MSc-IT

ADVANCED DATA STRUCTURE

GAUHATI UNIVERSITY
Institute of Distance and Open Learning

Semester- II

MSc-IT
Paper: INF 2056

Advanced Data Structure

www.idolgu.in

M.Sc.-IT-19-II-2056

GAUHATI UNIVERSITY
Institute of Distance and Open Learning

Second Semester

(under CBCS)

M.Sc.-IT

Paper: INF-2056

ADVANCED DATA STRUCTURE

Contents:

Contributors:

Mr. Ajit Das (Block I : Units- 1 & 4)
Asstt. Prof, Dept. of Computer
Science and Technology (Block II : Units- 4 & 5)
Bodoland University
Kokrajhar(BTAD), Assam

Dr. Swapnanil Gogoi (Block I: Units- 2 & 3)
Asstt. Prof., GUIDOL

Mr. Manash K. Gogoi (Block II: Units- 1, 2 & 3)
Asstt. Prof, Dept. of Computer Science
Handique Girls' College, Guwahati, Assam

Dr. Utpal Barman (Block III: Units- 1, 2, 3 & 4)
Asstt. Prof., Dept. of Computer
Science and Engineering
GIMT, Guwahati, Assam

Mrs. Pallavi Saikia (Block II: Unit- 6)
Asstt. Prof., GUIDOL

Content Editors:

Prof. Anjana Kakoti Mahanta
Head, Dept. of Computer Science, Gauhati University

Dr. Irani Hazarika
Asstt. Prof., Dept. of Computer Science, Gauhati University

Course Coordination:

Prof. Dandadhar Sarma Director, IDOL, Gauhati University
Prof. Anjana Kakoti Mahanta Prof., Dept. Computer Science, G.U.

Cover Page Designing:

Bhaskar Jyoti Goswami IDOL, Gauhati University

May, 2022

Published on behalf of Institute of Distance and Open Learning, Gauhati University by the Director, and printed at Gauhati University Press, Guwahati-781014.

BLOCK I:
REVIEW OF BASIC CONCEPTS IN DATA STRUCTURE

UNIT 1: INTRODUCTION TO DATA STRUCTURE

Unit Structure:

Space for learners:

1.1 INTRODUCTION

Since the beginning of computer invention, people are using the term data to describe the computer information. D**ata** is any set of characters that is transmitted and stored for some purpose, usually for analysis. Generally, for solving any problem by computer involves the use of data. Data should be arranged in some systematic way then it becomes meaningful. This meaningful data is called information. There are many ways by which we can organize data i.e. in structured form. To convert data in an appropriate structured form we need to know data structure.

Data structure defines a way of arranging all data items so that various operations can be performed on it in an effective way. The term data structure is used to describe the way data is stored. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency. For example, in linear search, data is not required to be sorted but in binary search technique, data is needed to be sorted. To develop a program from an algorithm we should select an appropriate data structure for that algorithm. So that we can represent data structure as:

Algorithm + Data structure = Program.

Data structures and algorithms both are inter-related to each other. An algorithm is a finite sequence of instructions, for solving a well-defined computational problem with a finite amount of effort in a finite length of time.

Space for learners:

1.2 UNIT OBJECTIVES

In this Chapter we will study the following concepts:

- Understand the basic concept of Data Structure
- Know about Algorithms and Abstract Data Type
- Array Traversal, insertion and deletion Operation.

1.3 CLASSIFICATION OF DATA STRUCTURES

1.3.1 Linear Data Structure

A **linear Data structure** is a type of data structure where data elements are arranged in sequential or in linear way where the elements are attached each other to its previous and next adjacent. Here, we can traverse all the elements in single run only. Linear data structures are easy to implement because computer memory is arranged in a linear way.

1.3.2 Non-linear Data Structure

A **non-linear data structures** is a type of data structure where data elements are not arranged sequentially or linearly. Here, we can't traverse all the elements in single run only. Non-linear data

structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure.

Space for learners:

1.3.3 Primitive Data Structures

Primitive data structures are the fundamental data structures that operated directly on the data and machine instructions as well. Sometimes we also called it as build-in data structures. Integers, character constants, floating point numbers, string constants and pointers come in this category.

1.3.4 Non-primitive Data Structures

Non-primitive or composite data structures is a user defined data structure that directly operate upon the machine instructions and is derived from the primitive data structures. The non-primitive or composite data structures emphasize on structuring from a group of homogenous or heterogeneous data items.

1.3.5 Differences Between Primitive and Non-primitive Data Structures

Primitive Data Structure	Non-Primitive Data Structure
Predefined in the language	Not defined in language and created by the programmer. Also called built-in.
Very easy to implement	Implementation is harder than Primitive data Structure.
Primitive Data structures will have a certain value	Non Primitive Data structure can have NULL value
The size totally depends upon the type of data structure	The size is not fixed
Can be used to call methods to perform operations	Cannot be used.
For example int, float, char, pointer etc.	For example Linked list, Tree, Graph, Stack, Queue etc.

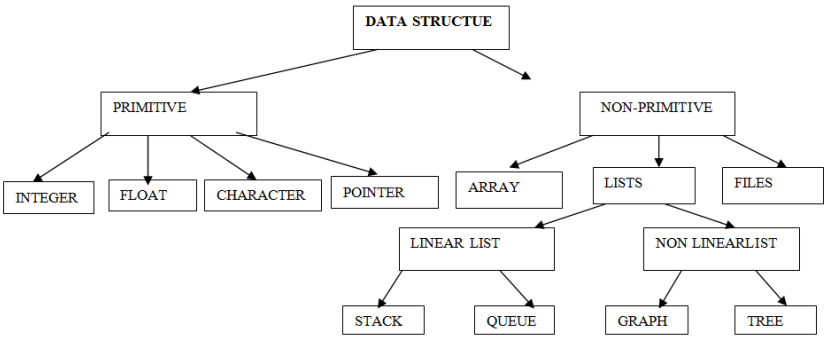


Figure 1: Types of data structure

1.4 WHAT IS AN ALGORITHM?

An algorithm is a procedure to accomplish a certain predefined task, exists some finite set of instructions or logic, written in order. Algorithm is just the core logic of a problem, which can be expressed either as **pseudocode** or by using a **flowchart**.

If an algorithm takes less time to execute and consumes less memory space then it is said to be efficient and fast. On the basis of following properties, the performance of an algorithm is measured in terms of **Space Complexity and Time Complexity**

1.4.1 Space Complexity

Space complexity is function represents the amount of memory space needed by the algorithm for implementation.

An algorithm requires space for following components:

- **Instruction Space:** It is required to store the executable program. Generally, this space is fixed, but it varies depending upon the number of lines of code in the program.
- **Data Space:** It is required to store all the constants and variables (including temporary variables) value.
- **Environment Space:** It is required to store the environment information needed to resume the suspended function.

Space for learners:

1.4.2 Time Complexity

Time Complexity of an algorithm is a function representing the amount of time required by the algorithm to be executed. Here *Time* means the number of comparisons between integers, the number of times is required to execute some inner loop is execute or some other natural unit related to the amount of real time the algorithm will take. So here our main motto is to try to keep the time required for implementing an algorithm should be in minimum time required.

Space for learners:

1.5 ABSTRACT DATA TYPE (ADT)

Abstract data type is a mathematical model or concept that is defined by a set of data and collection of operations that can be performed on that data. ADT only defines what operations are to be performed but not the implementation issues of operations. It will not clearly define how data will be organized in the memory and what types of algorithms can be used for implementing the operations. It gives an implementation as an independent view that is why it is called as "abstract". The ADT is made of with built-in data types, but operation logics are hidden. Some examples of ADT are Stack, Queue, List etc.

1.6 WHAT IS AN ARRAY?

An **Array structure** is an indexed collection of data items of the similar type. Indexed means array elements are numbered (starting at 0). The data type of the elements of the array may be any valid data types like int, float or char. The array element shares the same variable name but each element has a different index number, known as a subscript. Array elements are stored in consecutive memory cells. Every cell must be the same type.

Consider a situation where we are trying to store the age of 100 students. We can take an array variable name *age* of integer type. The individual elements of this array are-

age[0], age[1], age[2].....age[99]

i.e we can define the array like

int age[100];

Here *age* is an integer type array which consists of 100 elements.

The subscript starts from zero. So age[0] is the first element of the array, age[1] is the second element and so on.

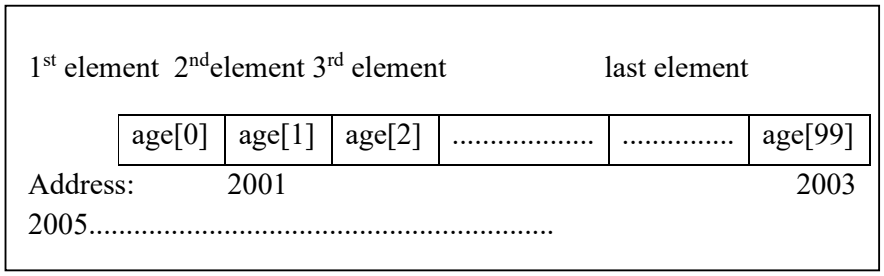


Figure 2: Structure of array elements

The following are some operations supported by an array.

- **Traverse** − Visiting every element in the array exactly once.
- **Search** − Search a particular element by using the given index or by the value in an array
- **Insertion** − Adds an element at the given index in an array.
- **Deletion** − Deletes an element at the given index in an array.

1.6.1 Traverse

Visiting every element in an array exactly once is termed as traversing the array.

A C program to traverses and prints the element of an array

```
#include<stdio.h>
void main()
{
   int a[100], n, i ;
   printf("Enter how many elements:\n");
   scanf("%d", &n);
   printf("Enter the elements of an array\n");
   for(i=0; i<n; i++)
   scanf("%d", &a[i]);
   printf("The elements in this array are:\n");
   for(i=0; i<n; i++)
   printf("Array[%d]= %d  \n", i, a[i]);
}
```

Space for learners:

OUTPUT

```
Enter how many elements:
5
Enter the elements of an array
3
6
7
9
3
The elements in this array are:
Array[0]= 3
Array[1]= 6
Array[2]= 7
Array[3]= 9
Array[4]= 3
```

Space for learners:

1.6.2 Searching Elements in an Array

Algorithm to search an element in an array:

The elements of an array can be searched using the following two methods-

1. Linear or sequential search
2. Binary search.

1.6.2.1 Linear or Sequential Search

In this method searching starts from the beginning of the list and continues till find the element or reaches the end of list. The searched item is compared with each element of the list one by one from the beginning.

Algorithm of Linear search:

- Start from the beginning element of array and one by one compare x(searched item) with each element of array
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

Following C program search elements in an array using Linear search method:

```
#include<stdio.h>
void main()
{
int a[100], n, i, item, flag = 0 ;
        printf("enter how many elements:\n");
        scanf("%d", &n);
        printf("enter the elements of an array\n");
        for(i=0; i<n; i++)
        scanf("%d", &a[i]);
        printf("The elements in this array are:\n");
        for(i=0; i<n; i++)
        printf(" Array[%d]= %d  \n", i, a[i]);
        printf("Enter  the searched item:\n");
        scanf("%d", &item );
        for(i=0; i<n; i++)
        {
if a[i]==item)
                {       flag=1;
                        break;
            }
   }
        if(flag==1)
        printf("%d is found in the array\n", item);
        else
        printf("%d is not found in the array\n", item);
        }
```

OUTPUT

Enter how many elements:

5

Enter the elements of an array

3

6

7

9

3

The elements in this array are:

Array[0]= 3

Array[1]= 6

Space for learners:

Array[2]= 7

Array[3]= 9

Array[4]= 3

Enter the searched item:

7

7 is found in the array

Space for learners:

1.6.2.2 Binary Search

In binary search the elements in the array have been already sorted. First compare the searched element with the middle element of the array. If item is found, search will stop, otherwise the array is divided into two halves. The first half contains the all the elements to the left side of the middle and other half contains the all the elements to the right side of the middle. As the array is already sorted, thus all the elements in the left side will be smaller than the middle and all of the elements in the right side will be the greater than the middle elements. Now if the searched element is less than the middle, it will search the elements in the left half portion otherwise it will search the elements in the right half portion. This process of comparing the elements with the middle elements and divide the array continues till the required item is found or get a portion where does not have any elements.

```
#include<stdio.h>
void main()
{
int a[100], n, i, item, flag = 0 ;
printf("enter how many elements:\n");
scanf("%d", &n);
printf("enter the elements of an array\n");
for(i=0; i<n; i++)
        scanf("%d", &a[i]);
printf("the elements in this array are:\n");
for(i=0; i<n; i++)
        printf(" array[%d]= %d    \n", i, a[i]);
printf("enter  the searched item:\n");
scanf("%d", &item );
flag=bsearch(a, n, item);
if(flag==-1)
```

```
        printf("%d is not found in the array\n", item);
else
        printf("%d is found at position %d in  the array\n", item,
flag);
}

intbsearch(int a[100],int n, int item)
{
int low=0, up=n-1, mid;
while(low<=up)
        {
        mid=(low+up)/2;
        if (item>a[mid])
                low=mid+1;
        else if(item<a[mid])
                up=mid-1;
        else
                return mid;
        }
return -1;
}
```

OUTPUT

Enter how many elements:

5

Enter the elements of an array

3

6

7

9

3

The elements in this array are:

Array[0]= 3

Array[1]= 6

Array[2]= 7

Space for learners:

Array[3]= 9

Array[4]= 3

Enter the searched item:

6

is found at position 2 in the array

Space for learners:

1.6.3 Insertion Operation of Array

In the array first location starts from 0 and the last element is less than the total size of an array. For insert an element in an array first we will put the location. We need to ensure that the location is available in the array or not. If the location where we need to put is not within the range of declared array, a warning message should be included. If it is within the range then we will shift each and every element to the next position one by one from that position or location.

A C program to insert an element to an array:

Suppose, *a* is an array, *n* is number of elements (size), *num* is a data element, *pos* is the location of the element to be inserted.

```
#include <stdio.h>
int main()
{
int a[100],n,pos,num,i;

printf("Enter the size of the array\n");
scanf("%d",&n);

printf("Enter the array\n");
for(i=0;i<n;i++)
   {
scanf("%d",&a[i]);
}

printf("\n");
printf("Enter the data you want to put\n");
scanf("%d",&num);
printf("Enter the position\n");
```

```
scanf("%d",&pos);

printf(" The original array is \n");
for(i=0;i<n;i++)
   {
printf("%d", a[i]);
printf("\t");
   }
/*---------- beginning of Insertion----------------*/
if(pos<=0|| pos>n+1)
   {
printf("Invalid position\n");
   }

else
for(i=n-1;i>pos;i--)
        {
a[i+1]=a[i];
        }

a[pos]=num;
n++;
/*---------- end of Insertion----------------*/

printf("\n Updated array is \n");
for(i=0;i<n;i++)
   {
printf("%d", a[i]);
printf("\t");
   }

return 0;
}
```

Space for learners:

OUTPUT

Enter the size of the array

6

Enter the array

3

4

5
6
7
8

Enter the data you want to put
1
Enter the position
3
The original array is
3 4 5 6 7 8
Updated array is
3 4 1 5 6 78

Space for learners:

1.6.4 Deletion in an array

We can delete a particular element from an array. If an element to be deleted is in i^{th} position then all elements from the $(i+1)^{th}$ position need to be shifted one step towards left. So $(i+1)^{th}$ element is copied to i^{th} location and $(i+2)^{th}$ to $(i+1)^{th}$ location and so on.

Algorithm to delete an element from an array:

a is an array

n is number of elements

pos is the location of the element to be deleted.

Deletion (a, n, pos)

Step 1: for i = pos to n-1 repeat step 2

Step 2: a[i] = a[i+1]

end for

Step 3: n = n -1

Implementation of Algorithm through C language:

```
#include <stdio.h>
int main()
{   int a[100],n,pos,i;

printf("Enter the size of the array\n");
scanf("%d",&n);
```

```
printf("Enter the array\n");
for(i=0;i<n;i++)
   {
scanf("%d",&a[i]);
   }
printf("The original array : \n");
for(i=0;i<=n-1;i++)
      {
printf("%d ",a[i]);
printf("\t");
      }

printf("\nEnter the position of element you want to delete :\n");
scanf("%d",&pos);
for(i=pos-1;i<=n-1;i++)
      {
a[i]=a[i+1];
}
n--;

printf(" Updated array is \n");
for(i=0;i<n;i++)
   {
printf("%d", a[i]);
printf("\t");
   }
return 0;
}
```

OUTPUT

```
Enter the size of the array
4
Enter the array
5
6
7
9
The original array:
5      6      7      9
Enter the position of element you want to delete:
```

Space for learners:

3
Updated array is
5 6 9

Space for learners:

CHECK YOUR PROGRESS

1. Multiple Choice Questions

(i) From below which one is non-linear type of data structure?
   a) Array
   b) Stack
   c) Queue
   d) Graph

(ii) From below which one is linear type data structure?
   a) Graph
   b) Tree
   c) AVL tree
   d) Queue.

(iii) From below which one is primitive data structure?
   a) Linked list
   b) Tree
   c) Pointer
   d) Graph

(iv) From below which one is non-primitive data structure?
   a) int
   b) Stack
   c) Float
   d)Char

(v) _____ is a mathematical model or concept that is defined by a set of data and collection of operations that can be performed on that data.
   a) Data Structure
   b) Abstract Data Type
   c) Algorithm
   d) Primitive data type

(vi) What are the advantages of arrays?
   a) Objects of mixed data types can be stored
   b) Elements in an array cannot be sorted
   c) Index of first element of an array is 1
   d) Easier to store elements of same data type

(vii) What are the disadvantages of arrays?

a) Data structure like queue or stack cannot be implemented

b) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size

c) Index value of an array can be negative

d) Elements are sequentially accessed

(viii) Which searching techniques not required elements to be sorted?

a) Linear search
b) Binary search
c) Interpolation search
d) All of the above.

(ix) In_____ we will shift each and every element to the next position one by one from location.

a) Insertion in array
b) Traverse in array
c) Deletion in array
d) Search in array.

(x) If an element to be _____ i^{th} position then all elements from the $(i+1)^{th}$ position need to be shifted one step towards left. So $(i+1)^{th}$ element is copied to i^{th} location and $(i+2)^{th}$ to $(i+1)^{th}$ location and so on.

a) Inserted
b) Traverse
c) Search
d) Deleted

2. Fill in the blanks

(i) Integers, character constants, floating point numbers, string constants and pointers are_____ type of Data Structure.

(ii) Linked list, Tree, Graph, Stack, Queue are _____ type of Data Structure.

(iii) Space complexity is function represents the amount of _____space needed the algorithm for implementation.

(iv) Time Complexity of an algorithm is a function representing the amount of _____required by the algorithm to be executed.

(v) Array elements are stored in _____memory cells.

Space for learners:

(vi) In binary search the elements in the array need to be _____ .
(vii) In deletion an element from an array we need to be shifted all the element start from the next element which to be deleted one step towards_____.

Space for learners:

1.7 SUMMING UP

- **Data** is any set of characters that is transmitted and stored for some purpose, usually for analysis.
- Data structure defines a way of arranging all data items so that various operations can be performed on it in an effective way. We can represent data structure as: Algorithm + Data structure = Program.
- A **linear Data structure** is a type of data structure where data elements are arranged in sequential or in linear way where the elements are attached each other to its previous and next adjacent.
- A **non-linear data structures** is a type of data structure where data elements are not arranged sequentially or linearly.
- **Primitive data** structures are the fundamental data structures that operated directly on the data and machine instructions as well. Integers, character constants, floating point numbers, string constants and pointers comes under this category.
- **Non-primitive or composite data structures** is a user defined data structure that directly operate upon the machine instructions and is derived from the primitive data structures.
- An **algorithm** is a step by step procedure to accomplish a certain predefined task, exists some finite set of instructions or logic, written in order.
- **Space complexity** is function represents the amount of memory space needed the algorithm for implementation.
- **Time Complexity** of an algorithm is a function representing the amount of time required by the algorithm to be executed. Here *Time* means the number of comparisons between integers, the number of times is required to execute some inner loop is

execute or some other natural unit related to the amount of real time the algorithm will take.

- **Abstract data type** is a mathematical model or concept that is defined by a set of data and collection of operations that can be performed on that data. The ADT is made of with built-in data types, but operation logics are hidden.
- An **Array data structure** is an indexed collection of data items of the similar type.**Indexed** means that the array elements are numbered (starting at 0).
- The following are some functions supported by an array.
  - **Traverse** − Visiting every elements in the array exactly once.
  - **Search** − Search an particular element by using the given index or by the value in an array
  - **Insertion** − Adds an element at the given index in an array.
  - **Deletion** − Deletes an element at the given index in an array.
- Two important searching method which can be implemented by using array-
  - Linear or sequential search
  - Binary search.
- In **linear searching method**, searching starts from the beginning of the list and continues till find the element or reaches the end of list. The searched item is compared with each elements of the list one by one from the beginning.
- In **binary search**, the elements in the array need to be sorted. First compare the searched element with the middle element of the array. If item is found, search will stop, otherwise the array is divided into two halves, the first half contains the all the elements left side from the middle and other half contains the all the elements to the right side of the middle. Now all the elements in the left side will be smaller than the middle and all the right side elements will be the greater than the middle elements. Now if the searched elements is less than the middle, it will search the elements in the left half portion otherwise it

Space for learners:

will search the elements in the right half portion. This process of comparing the elements with the middle elements and divide the array continues till the required item is found or get a portion where does not have any elements.

- In **insertion in an array** we will shift each and every element to the next position one by one from insert element position.
- In **deletion an element from an array** we need to be shifted all the element start from the next element which to be deleted one step towards left.

Space for learners:

1.8 ANSWERS TO CHECK YOUR PROGRESS

1. (i) (d) , (ii) (d), (iii) (c), (iv) (b) , (v) (b) , (vi) (d), (vii) (b) , (viii) (a) , (ix) (a), (x) (d)
2. (i) Primitive, (ii) Non-Primitive, (iii) Memory, (iv) Time, (v) Consecutive, (vi) Sorted, (vii) Left

1.9 POSSIBLE QUESTIONS

1. What are the advantages and disadvantages of arrays?
2. What is Data Structure?
3. What do you mean by an Algorithm
4. What is Pseudocode?
5. What is Flowchart?
6. Explain Primitive Data Structure
7. Explain Non-Primitive Data Structure.
8. What is space complexity?
9. What is Time complexity?
10. What is Abstract Data type?
11. What is an Array?
12. Explain the differences between Primitive and No-primitive data structure.
13. Write a C program to implement the Linear Search Algorithm.

14. Write the algorithm of Binary Search.
15. Explain the procedure of insertion an element in any position in an array.
16. Explain the procedure of deletion an element in any position from an array.

Space for learners:

1.10 REFERENCES AND SUGGESTED READINGS

- Srivastava, Suresh Kumar, and Deepali Srivastava. *Data Structures through C in depth*. BPB publications, 2004.
- Thareja, Reema. *Data structures using C*. Oxford University Press, Inc., 2011.

---×---

UNIT 2: LINKED LIST

Unit Structure:

Space for learners:

2.1 INTRODUCTION

In unit 1, basics of data structure has been discussed. We have learnt about array from this unit. As a static data structure, array has some limitations. In this unit, we are going to learn about linked list which is a dynamic data structure. Different types of linked list and operations on these linked lists will be discussed in this unit.A comparison between linked list and array has also been provided in this unit so that the advantages and disadvantage of linked list over array can be explored.

2.2 UNIT OBJECTIVES

After reading this unit you are expected to be able to learn:

- The basic concepts of linked list.
- Difference between array and linked list.
- About different types of linked lists.
- About three important operations (Insertion, Deletion and Traversal) on Linear linked list, Doubly linked list and Circular linked list.
- Implementations of the mentioned operations on Linear linked list, Doubly linked list and Circular linked list using C++ programming.
- About Doubly circular linked list and its implementation using C++ programming.

Space for learners:

2.3 BASICS OF LINKED LIST

Linked list is a linear data structure. It is also termed as dynamic data structure because the size of the structure can be increased or decreased as per the requirement of the user at run-time. A linked list can be defined as a list of nodes where each node contains data and memory addresses that point to its connected nodes. The number of memory addresses stored in a node is dependent upon the type of a particular linked list. It may be one or two. In case of linked list, data may not be stored in contiguous memory locations.

2.3.1 Types of Linked List

Linked lists are categorized into three basic types that are mentioned as follows.

1) Singly linked list or Linear linked list
2) Doubly linked list
3) Circular linked list

A fourth type of linked list can be developed by combining the concept of Doubly linked list and Circular linked list. This type of linked list can be termed as Doubly circular linked list.

Space for learners:

2.3.2 Comparison of Linked List and Array

In this section, some important points are provided to compare linked list and array. This comparison will help us to understand the advantages and disadvantages of linked list over array. Now let us try to understand the points available in the following table.

Array	Linked List
The size of an array is determined at compile-time and it cannot be altered at run-time.	The size of a linked list can be increased or decreased at run-time. When a new node is added to a linked list then its size is increased. Alternatively if an old node is removed from a linked list then its size is decreased.
If the array size of an array is smaller than the number of data that are required to be stored then it is not possible to store all the data into the array.	Any number of data can be added into a linked list at run-time depending upon the availability of memory locations.
If the array size of an array is much bigger than the number of data that are required to be stored then most of the memory spaces in the array are not utilized. In that case, lots of memory wastages are occurred.	As insertion of new nodes and deletion of existing nodes are performed at run-time in linked list, so there is no memory wastage happened.
In some cases, insertion of new data into an array and deletion of old data from an array require data relocation which is a very time consuming process.	In case of linked list, insertion of new node and deletion of existing node can be performed at any position in the list without performing any node shifting operation.
Data are stored in contiguous memory locations in array. So an array doesn't require storing	Data may not be stored in contiguous memory locations in linked list. So each node in a

memory address of the next data for data traversal operation.	linked list must store at least the memory address of the next node so that traversal from one node to its next node can be performed. As a result, more memory locations are required to implement linked list than array.
Random access or direct access of data is possible in array. We can access any data in an array directly by using the subscript value of the particular data with the array name.	Random access or direct access of data in linked list is not possible because in case of linked list, data may not be stored in contiguous memory locations. In linked list, a particular data can be accessed by traversing to its node with the help of the pointer to the next node stored in each node. As a result, more time is required to access data in linked list than array.

Space for learners:

2.3.3 Applications of Linked List

Linked list is applied in various implementations. Some of the most common applications of linked list are presented as follows.

1) Linked list is used to represent graph in memory.
2) Linked list is used to implement dynamic stacks and dynamic queues.
3) Linked list is used in the implementations of tree data structures and heaps.
4) Linked list is used to prevent hash collision in hashing.
5) Linked list is used in dynamic memory allocation to keep track of free memory blocks.
6) Linked list can also be used to store and process polynomials.

2.4 SINGLY LINKED LIST

Space for learners:

Singly linked list is the most basic type of linked list. It is a collection of nodes where each node contains two fields that are data field and address field(Figure 2.1).The data field can contain data and the address field can contain the memory address of the next node in the list. The address field of the last node contains NULL. NULL is value which means the pointer that stores NULL points nothing. In general, one special pointer is used to store the memory address of the first node in a singly linked list and this pointer can be used to perform different operations on the list like Insertion, Deletion, and Traversing etc. A diagrammatical representation of a singly linked list is presented in Figure 2.2. The Singly linked list as shown in this figure, consist of three nodes where the memory address of the first node is 411 that is stored in the special pointer, 'Start'.

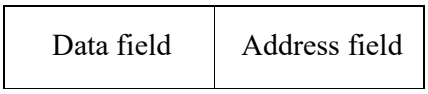


Figure 2.1: Diagrammatic representation of a singly linked list node

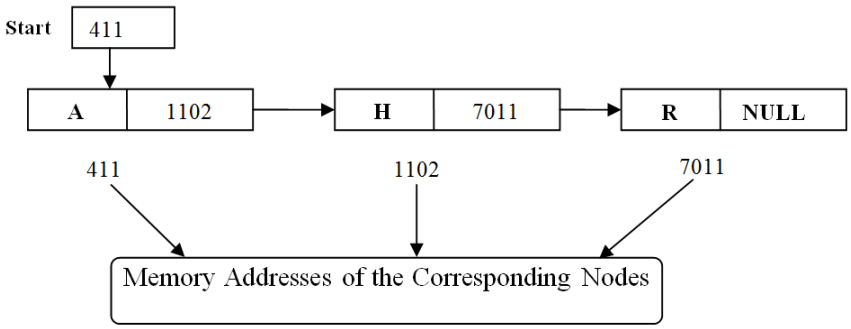


Figure 2.2: Diagrammatic representation of a singly linked list

2.4.1 Insert Operation on Singly Linked List

In this section, insertion of a new node to a singly linked list will be discussed and implemented using C++ programming. Let us consider 'Start' as the name of the special pointer which stores the memory address of the first node in the list. To insert a new node, at first, a node has to be created by allocating memory at run-time.

After memory allocation, the address of the created node is stored in a pointer. Let us consider 'TempPtr' as the name of this pointer. Then appropriate data is assigned to the data field and NULL value is assigned to the address field of the newly created node.

Space for learners:

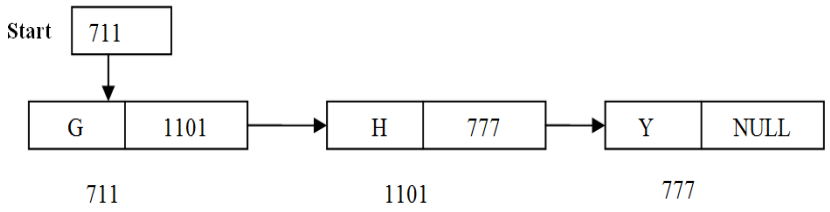


Figure 2.3:A singly linked list with three existing nodes

Insertion of a new node at the first position in a singly linked list:

If a singly linked list doesn't contain any node then NULL is stored in Start. In that case, to insert a new node, the memory address stored in TempPtr is assigned to Start. Otherwise let us try to observe Figure 2.4 to understand the process of inserting a new node at first position in a singly linked list.

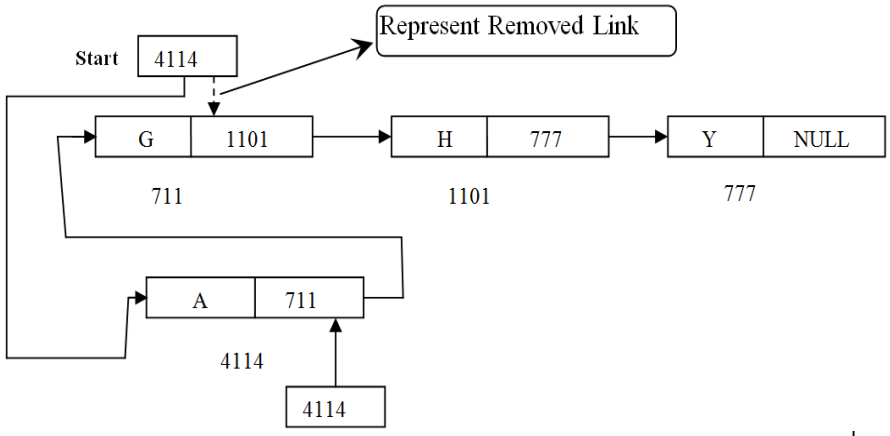


Figure 2.4: Singly linked list after insertion of a new node at first position

Figure 2.4 represents the singly linked list after inserting a new node at the first position in the Singly linked list that is represented in Figure 2.3. In this insertion operation, two steps have been

performed. At first the memory address of the current first node is assigned to the address field of the newly created node so that the current first node is linked to the new node as next node. Then in the second step, the memory address of the newly created node is assigned to Start so that it is linked to the list as the new first node.

Insertion of a new node at the last position in a singly linked list:

If a singly linked list doesn't contain any node then the memory address stored in TempPtr is assigned to Start. Otherwise let us try to observe Figure 2.5 to understand the process of inserting a new node at last position in a Singly linked list.

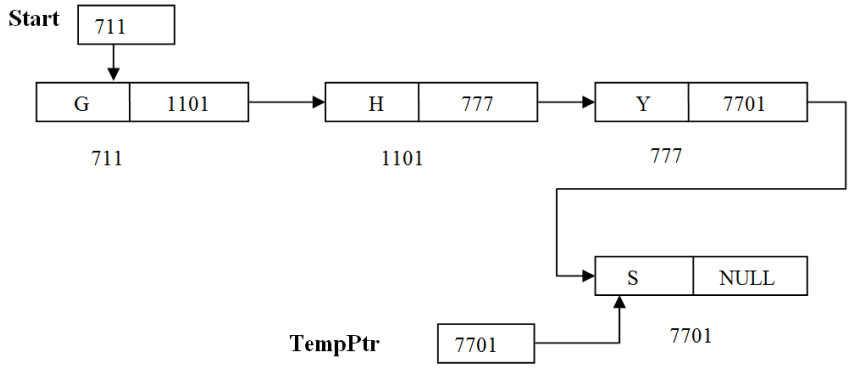


Figure 2.5:Singly linked list after insertion of a new node at last position

Figure 2.5 represents the singly linked list after inserting a new node at the last position in the singly linked list that is represented in Figure 2.3. In this insertion operation, two steps have been performed. At first, the last node has to be reached by visiting all the nodes from the first node in the list using a pointer. Then in the second step, the NULL value stored in the address field of the current last node of the list is replaced by the memory address of the new node. As a result, the new node is inserted at the last position in the linked list.

Insertion of a new node at a specific position in a singly linked list:

To insert a new node at a specific position, at first the specific position has to be read. If a list doesn't contain any node then the

Space for learners:

input value of the specific position must be one otherwise it will be invalid. Again, if a list contains N number of nodes then the input value of the specific position cannot be greater than N+1. Let us try to observe Figure 2.6 to understand the process of inserting a new node at a specific position that is 3 in the mentioned figure.

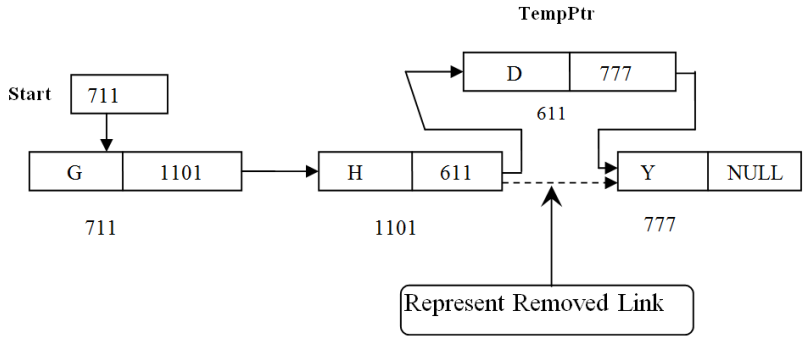


Figure 2.6:Singly linked list after insertion of a new node at 3rdposition

Figure 2.6 represents the singly linked list after inserting a new node at 3rd position in the singly linked list that is represented in Figure 2.3. In this process of insertion operation, at first, the second node has to be reached using a pointer. In the second step, the memory address stored in the address field of the second node is assigned to the address field of the newly created node. Finally, the memory address of the new node is assigned to the address field of the second node. As a result, the new node is inserted at the 3rd position in the list.

Program 2.1:C++ Program to insert a new node to a Singly Linked List

```
#include<iostream.h>
#include<conio.h>

struct Node     // User defined data type to create Nodes
{
        char data;
        struct Node *next;
};

typedef struct Node Node;
```

Space for learners:

Space for learners:

```
class Singly_List
{
        private:
                  Node *Start;
                int S_Pos;
        public:
                  Singly_List()
                  {
                     Start=NULL;
                  }
                void Insert_At_First();
                void Insert_At_Last();
                int Insert_At_Specific();
                void Display_List();
};

void Singly_List::Insert_At_First()// Function to insert node at first
                                                    position
{
        Node *TempPtr;
        TempPtr= new Node;
        TempPtr->next= NULL;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(Start==NULL)
          Start=TempPtr;
        else
        {
        TempPtr->next=Start;
        Start=TempPtr;
        }
}

void Singly_List::Insert_At_Last()  // Function to insert node at Last
position
{
        Node *TempPtr,*PtrLast;
        TempPtr= new Node;
        TempPtr->next= NULL;
        cout<<"\n Enter a character=";
```

```
        cin>>TempPtr->data;
        if(Start==NULL)
         Start=TempPtr;
        else
        {
        PtrLast=Start;
        while(PtrLast->next!=NULL)
                PtrLast=PtrLast->next;
        PtrLast->next=TempPtr;
        }
}

int Singly_List::Insert_At_Specific()   // Function to insert node at a
specific position
{
        Node *TempPtr,*PtrPrev;
        int count=1;
        TempPtr= new Node;
        TempPtr->next= NULL;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        cout<<"\n Enter the value for new node position=";
        cin>>S_Pos;
        if(Start==NULL)
        {
        if(S_Pos==1)
          {
            Start=TempPtr;
        return(S_Pos);
         }
        else
        {
        cout<<"\n Invalid input value for new node position";
        return(0);
          }
        }
        else
        {
        if(S_Pos==1)
        {
        TempPtr->next=Start;
```

Space for learners:

```
        Start=TempPtr;
        return(S_Pos);
        }
        else
        {
         PtrPrev=Start;
        while(PtrPrev->next!=NULL && count< S_Pos-1)
        {
                count=count+1;
                 PtrPrev=PtrPrev->next;
         }
        if(count==S_Pos-1)
        {
                TempPtr->next=PtrPrev->next;
                PtrPrev->next=TempPtr;
                return(S_Pos);
          }
        else
          {
                cout<<"\n Invalid input value for new node position";
                return(0);
        }
          }
    }
}

void Singly_List::Display_List() //Function to display the linked list
{
   Node *TempPtr;
   TempPtr=Start;
if(Start==NULL)
        cout<<"\n Empty List";
else
   {
cout<<"\n Data available in the list are:\n";
while(TempPtr!=NULL)
{
        cout<<TempPtr->data;
        cout<<"\t";
         TempPtr=TempPtr->next;
}
```

Space for learners:

```
}
}

int main()
{
Singly_List SL1;
char more='y';
int choice,temp;
clrscr();
while(more=='y' || more=='Y')
 {
        cout<<"\n 1. Insert as First Node";
        cout<<"\n 2. Insert as Last Node";
        cout<<"\n 3. Insert at a Specific Position";
        cout<<"\n Enter your choice=";
        cin>>choice;
        switch(choice)
         {
                case 1: SL1.Insert_At_First();
                        cout<<"\n After Insertion::";
                        SL1.Display_List();
                        break;
                case 2: SL1.Insert_At_Last();
                        cout<<"\n After Insertion::";
                        SL1.Display_List();
                        break;
                case 3: temp =SL1.Insert_At_Specific();
                        if(status==0)
                        cout<<"\nInsertion Unsuccessful";
                        else
                        {
                        cout<<"\n After Insertion::";
                        SL1.Display_List();
                        }
                        break;
                default: cout<<"\n Invalid input for your choice";
         }

        cout<<"\n Input 'y' or 'Y' to insert one more node=";
        cin>>more;
   }
```

Space for learners:

```
getch();
return 0;
}
```

Space for learners:

2.4.2 Delete Operation on Singly Linked List

In this section, the deletion operation on Singly linked list will be discussed and implemented using C++ programming. Let us consider 'Start' as the name of the special pointer which stores the memory address of the first node in the list.

Deletion of the first node of a singly linked list:

To delete the first node, at first, the memory address of the first node that is stored in Start has to be assigned to a pointer. Let us consider the name of this pointer is DeletePtr. Then in the second step, the memory address stored in the address field of the first node is assigned to Start so that the second node becomes first node in the list. Finally, by using the memory address stored in DeletePtr, the memory allocated for the earlier first node is released.

Figure 2.7 represents the singly linked list after deleting the first node from the list that is represented in Figure 2.3.

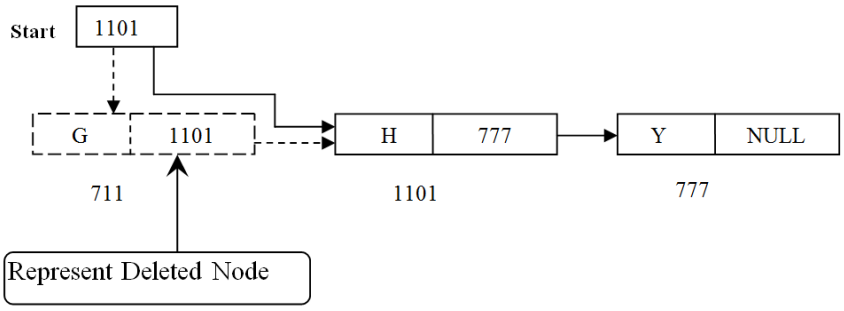


Figure 2.7: Singly linked list after deletion of the first node

Deletion of the Last node of a singly linked list:

If the address field of the first node in a singly linked list contains NULL then it means that there is only one node available in that list and that node is the first node as well as also the last node. In that

case, using the memory address stored in Start, the memory allocated for the last node can be released to delete the last node and then NULL value must be assigned to Start.

If the list contains more than one node then to delete the last node, at first, previous node to the last node has to be reached from the first node using a pointer. Then this pointer can be used to access the address field of that node so that the memory address of the last node can be accessed. In the third step, the memory address of the last node is assigned to a pointer. Let us consider the name of this pointer is DeletePtr. Using DeletePtr, the memory allocated for the last node is released to delete the last node. Finally, NULL value is assigned to the address field of the previous node to the earlier last node so that now it becomes the last node in the list.

Figure 2.8 represents the singly linked list after deleting the last node from the list that is represented in Figure 2.3.

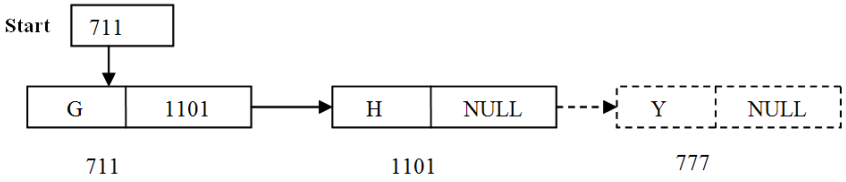


Figure 2.8:Singly linked list after deletion of the lastnode

Deletion of a Node Available in a Specific Position in a Singly linked list:

To delete a node that is available in a specific position in a singly linked list, at first, the value of the specific position has to be read. Now if the list contains N numbers of nodes then the input value of the specific position cannot be greater than N. In this deletion operation, two pointers are required. Let us consider these as PtrPrev and PtrPos where PtrPos will be used to point the node available at the specific position and PtrPrev will be used to point its previous node.Then the memory address of the next node to the node pointed by PtrPos is accessed by using PtrPos and it is assigned to the address field of the node that is pointed by PtrPrev. Finally, using PtrPos, the memory allocated by the node available in the specific position is released.

Space for learners:

Figure 2.9 represents the singly linked list after deleting the 2nd node from the list that is represented in figure 2.3.

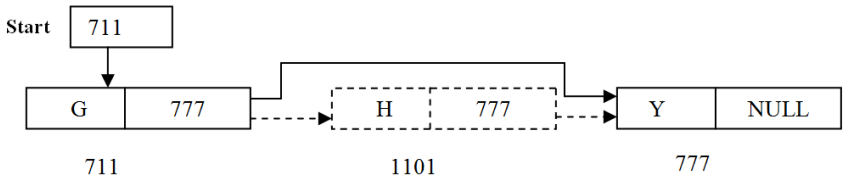


Figure 2.9:Singly linked list after deletion of the 2nd node

Program 2.2: C++ program to delete an existing node from a singly linked list

```
#include<iostream.h>
#include<conio.h>

struct Node     // User defined data type to create Nodes
{
        char data;
        struct Node *next;
};

typedef struct Node Node;

class Singly_List
{
        private:
                Node *Start;
              int S_Pos,i;
        public:
                Singly_List()
                {
                   Start=NULL;
                }
              void Create_List();
              void Remove_First_Node();
              void Remove_Last_Node();
              int Remove_Specific();
              void Display_List();
};
```

Space for learners:

Space for learners:

```
void Singly_List::Create_List()   // Function to create a Singly linked list
{
        Node *TempPtr;
        char more='y';
        i =1;
        while(more=='y'||more=='Y')
        {
        cout<<"\n Insert "<< i <<"th Node::";
        TempPtr=new Node;
         TempPtr->next=NULL;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(Start==NULL)
                 Start=TempPtr;
        else
        {
                TempPtr->next=Start;
                Start=TempPtr;
        }
        i++;
        cout<<"\n Enter 'y' or 'Y' to add one more node=";
        cin>>more;
        }
}

void Singly_List::Remove_First_Node() // Function to remove the first node
{
        Node *DeletePtr;

        if(Start==NULL)
        cout<<"\n Empty linked list";
        else
        {
        DeletePtr=Start;
        Start=Start->next;
        delete DeletePtr;
        }
}
```

Space for learners:

```
void Singly_List::Remove_Last_Node()  // Function to remove the last node
{
        Node *DeletePtr,*PtrPrev;

        if(Start==NULL)
        {
        cout<<"\n Empty linked list";
        }
        else
        {
        DeletePtr=Start;
        if(Start->next == NULL)
        Start= NULL;

        else
          {
                while(DeletePtr->next!=NULL)
                {
                        PtrPrev=DeletePtr;
                        DeletePtr=DeletePtr->next;
                }
                PtrPrev->next=NULL;
          }
        delete DeletePtr;
        }
}

int Singly_List::Remove_Specific()  /* Function to remove the node available at a specific position */
{
        Node *DeletePtr,*PtrPrev;
        int count=1;

        cout<<"\n Enter the value of the node position=";
        cin>>S_Pos;
        if(Start==NULL)
        {
        cout<<"\n Empty linked list";
        return(0);
```

```
        }
        else
        {
        DeletePtr=Start;
        if(S_Pos==1)
        {
        Start=Start->next;
        delete DeletePtr;
        return(S_Pos);
        }
        else
         {
        while(DeletePtr->next!=NULL && count<S_Pos)
        {
                count=count+1;
                 PtrPrev=DeletePtr;
                DeletePtr=DeletePtr->next;
            }
        if(count==S_Pos)
            {
                  PtrPrev->next=DeletePtr->next;
                delete DeletePtr;
                return(S_Pos);
            }
        else
            {
                cout<<"\n Invalid input value for the node position";
                return(0);
            }
          }
    }
}

void Singly_List::Display_List() //Function to display the linked list
{
   Node *TempPtr;
   TempPtr=Start;
if(Start==NULL)
        cout<<"\n Empty List";
else
   {
```

Space for learners:

```
cout<<"\n Data available in the list are:\n";
while(TempPtr!=NULL)
 {
          cout<<TempPtr->data;
          cout<<"\t";
            TempPtr=TempPtr->next;
}
}
}

int main()
{
Singly_List SL1;
char more='y';
int choice,temp;
clrscr();
cout<<"\n Create a Singly Linked List";
 SL1.Create_List();
 SL1.Display_List();
while(more=='y' || more=='Y')
{
          cout<<"\n 1. Delete the First Node";
          cout<<"\n 2. Delete the Last Node";
          cout<<"\n 3. Delete the Node at Specific Position";
          cout<<"\n Enter your choice=";
          cin>>choice;
          switch(choice)
           {
                    case 1: SL1.Remove_First_Node();
                              cout<<"\n After Deletion of the First Node::";
                              SL1.Display_List();
                              break;
                    case 2: SL1.Remove_Last_Node();
                              cout<<"\n After Deletion of the Last Node::";
                              SL1.Display_List();
                              break;
                    case 3: temp=SL1.Remove_Specific();
                              if(temp==0)
                              cout<<"\nDeletion Unsuccessful";
                              else
                              {
```

Space for learners:

```
                        cout<<"\n After Deletion of the Node at "<<temp<<"th position::";
                            SL1.Display_List();
                        }
                        break;
                default: cout<<"\n Invalid input for your choice";
            }

        cout<<"\n Input 'y' or 'Y' to insert one more node=";
        cin>>more;
    }
getch();
return 0;
}
```

Space for learners:

2.4.3Traversal Operation on Singly Linked List

The traversal operation can be performed in a singly linked list by moving from one node to its next node with the help of the memory address stored in the address field of the node. The traversal of nodes is started with the first node and it is continued to the last node. The memory address of the first node can be obtained from the special pointer associated with the list. Then by accessing the address fields of each node, the traversal operation can be performed and it stops when NULL value is encountered as the last node contains NULL value in its address field. As each node in a singly linked list contains the memory address of its next node, so traversal can be performed in forward directions only.

STOP TO CONSIDER

Backward traversal (Last node to first node) is not possible in singly linked list.

2.5 DOUBLY LINKED LIST

Doubly linked list is a linear list of nodes where each node contains three fields. Among these three fields, one is data field and other two are address fields (Figure 2.10). The data field stores the data. On the other hand, the two address fields of a node are used to store the memory addresses of its previous and next node available in the list.

One address field of the first node stores NULL because there is no previous node available to the first node in a doubly linked list the list. Similarly, one address field of the last node stores NULL because there is no next node available to the last node in a doubly linked list. In general, one special pointer is used to store the memory address of the first node available in a particular doubly linked list. This pointer can be used to perform different operations on the doubly linked list. A diagrammatic representation of a doubly linked list is presented in the Figure 2.11 where 'Dstart' is the special pointer that stores the memory address of the first node in the list.

If we compare doubly linked list with Singly linked list then it is observed that more memory space is required to implement a doubly linked list than the Singly linked list. It happens because each node in a doubly linked list contains an extra address field to store the memory address of its previous node. But due to this extra address field in each node, data traversal can be performed in both directions (forward and backward) in a doubly linked list. On the other hand, we have already learnt that data traversal can be performed only in forward direction in a singly linked list.

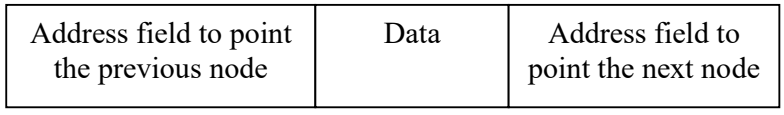


Figure 2.10: Diagrammatic representation of a doubly linked list node

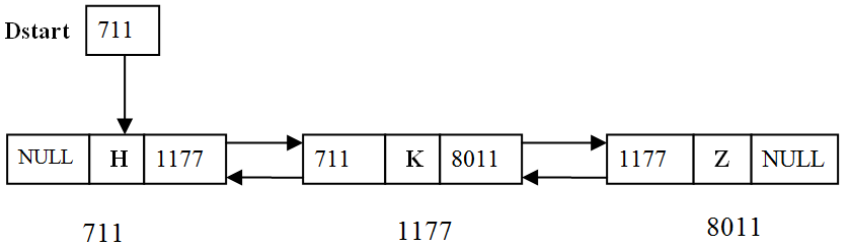


Figure 2.11:Diagrammatic representation of a doubly linked list

STOP TO CONSIDER
Any node can be reached from any node in a doubly linked list.

Space for learners:

2.5.1 Insert Operation on Doubly Linked List

Space for learners:

In this section, insertion of a new node to a doubly linked list will be discussed and implemented using C++ programming. Let us consider 'Dstart' as the name of the special pointer which stores the memory address of the first node in the list. To insert a new node, at first, a node has to be created by allocating memory at run-time. After memory allocation, the address of the created node is stored in a pointer. Let us consider 'TempPtr' as the name of this pointer. Then appropriate data is assigned to the data field and NULL value is assigned to both address fields of the newly created node.

Insertion of a new node at the first position in a doubly linked list:

If a Doubly linked list doesn't contain any node then NULL is stored in Dstart. In that case, to insert a new node, the memory address stored in TempPtr is assigned to Dstart. Otherwise let us try to observe Figure 2.12 to understand the process of inserting a new node at first position in a doubly linked list.

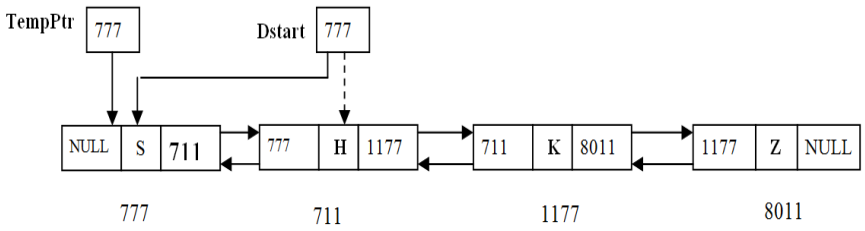


Figure 2.12: Doubly linked list after insertion of a new node at the first position

Figure 2.12 represents the doubly linked list after inserting a new node at the first position in the doubly linked list that is represented in Figure 2.11. In this insertion operation, three steps have been performed presented as follows.

1) At first, the memory address of the current first node is assigned to one address field of the newly created node so that the current first node is linked to the new node as next node.
2) In the second step, the memory address of the newly created node is assigned to the address field of the current first node

which contains NULL value. As a result, the new node is linked to the current first node as previous node.

3) Finally, the memory address of the new node is assigned to Dstart so that it is linked to the list as the new first node.

Insertion of a new node at the last position in a doubly linked list:

If a doubly linked list doesn't contain any node then the memory address stored in TempPtr is assigned to Dstart. Otherwise let us try to observe Figure 2.13 to understand the process of inserting a new node at last position in a Doubly linked list.

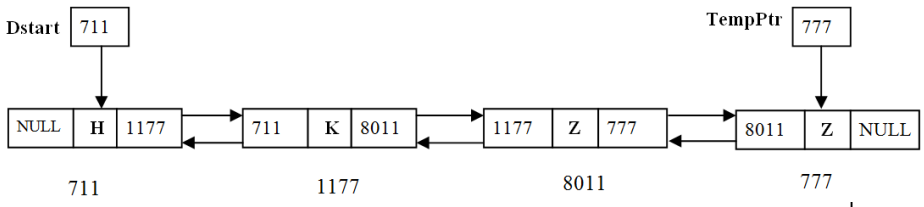


Figure 2.13: Doubly linked list after insertion of a new node at the last position

Figure 2.13 represents the doubly linked list after inserting a new node at the last position in the doubly linked list that is represented in figure 2.11. In this insertion operation, three steps have been performed presented as follows.

1) At first, the current last node has to be reached by visiting all the nodes from the first node in the list using a pointer.
2) In the second step, the NULL value stored in one address field of the current last node is replaced by the memory address of the new node. As a result, the new node is linked to the current last node as next node.
3) Finally, the memory address of the current last node is assigned to one address field of the new node so that the current last node is linked to the new node as previous node.

Space for learners:

Insertion of a new node at a specific position in a Doubly linked list:

To insert a new node at a specific position, at first the specific position has to be read. If a list doesn't contain any node then the input value of the specific position must be one otherwise it will be invalid. Again, if a list contains N number of nodes then the input value of the specific position cannot be greater than N+1. Let us try to observe Figure 2.14 to understand the process of inserting a new node at a specific position that is 3 in the mentioned figure.

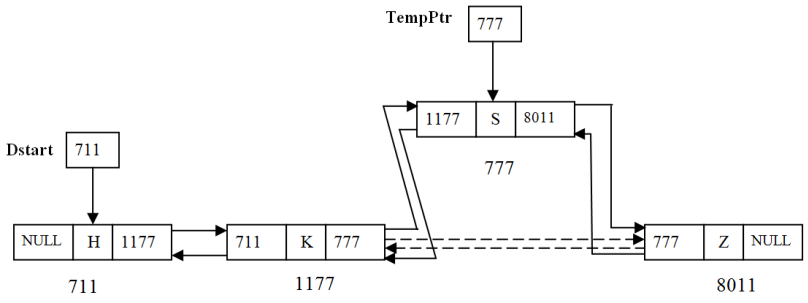


Figure 2.14: Doubly linked list after insertion of a new node at the 3rd position

Figure 2.14 represents the doubly linked list after inserting a new node at 3rd position in the doubly linked list that is represented in Figure 2.11. The steps performed in this process of insertion operation are presented as follows.

1) At first, the second node has to be reached using a pointer.
2) In the second step, the memory address of the next node to the second node is assigned to one address field of the new node so that the next node to the second node is also linked to the new node as next node.
3) In the third step, the second node is linked to the new node as previous node by assigning the memory address of the second node to the other address field of the new node.
4) In the fourth step, the memory address of the second node stored in one address field of its next node is replaced by the memory address of the new node. As a result, the new node becomes the previous node to the next node of the second node.

Space for learners:

5) In the final step, the new node is linked to the second node as next node by assigning the memory address of the new node to one address field of the second node.

Space for learners:

Program 2.3: C++ program to insert new node to a doubly linked list

```
#include<iostream.h>
#include<conio.h>

struct Node     // User defined data type to create Nodes
{
        char data;
        struct Node *next; //Pointer to point the next node
        struct Node *prev; //Pointer to point the previous node
};

typedef struct Node Node;

class Doubly_List
{
        private:
                Node *Dstart;
                int S_Pos;
        public:
                Doubly_List()
                {
                Dstart=NULL;
                }
                void Insert_At_First();
                void Insert_At_Last();
                int Insert_At_Specific();
                void Display_List();
};

void Doubly_List::Insert_At_First() // Function to insert node at first position
{
        Node *TempPtr;
        TempPtr= new Node;
        TempPtr->next= NULL;
```

```
        TempPtr->prev= NULL;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(Dstart==NULL)
         Dstart=TempPtr;
        else
        {
         TempPtr->next=Dstart;
          Dstart->prev=TempPtr;
         Dstart=TempPtr;
        }
}

void Doubly_List::Insert_At_Last()  // Function to insert node at
Last position
{
        Node *TempPtr,*PtrLast;
        TempPtr= new Node;
TempPtr->next= NULL;
        TempPtr->prev= NULL;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
if(Dstart==NULL)
         Dstart=TempPtr;
else
        {
        PtrLast=Dstart;
        while(PtrLast->next!=NULL)
                PtrLast=PtrLast->next;
        PtrLast->next=TempPtr;
         TempPtr->prev=PtrLast;
        }
}

int Doubly_List::Insert_At_Specific()   // Function to insert node at a
specific position
{
        Node *TempPtr,*PtrPrev;
        int count=1;
        TempPtr= new Node;
        TempPtr->next= NULL;
```

Space for learners:

```
TempPtr->prev= NULL;
cout<<"\n Enter a character=";
cin>>TempPtr->data;
cout<<"\n Enter the value for new node position=";
cin>>S_Pos;
if(Dstart==NULL)
{
if(S_Pos==1)
{
Dstart=TempPtr;
return(S_Pos);
}
else
{
cout<<"\n Invalid input value for new node position";
return(0);
}
}
else
{
if(S_Pos==1)
{
TempPtr->next=Dstart;
Dstart->prev=TempPtr;
 Dstart=TempPtr;
return(S_Pos);
}
else
{
PtrPrev=Dstart;
while(PtrPrev->next!=NULL && count<S_Pos-1)
 {
          count=count+1;
           PtrPrev=PtrPrev->next;
}
if(count==S_Pos-1)
{
          if(PtrPrev->next==NULL)
          {
           PtrPrev->next=TempPtr;
          TempPtr->prev=PtrPrev;
```

Space for learners:

```
                }
                else
                {
                        TempPtr->next=PtrPrev->next;
                        TempPtr->prev=PtrPrev;
                        (PtrPrev->next)->prev=TempPtr;
                        PtrPrev->next=TempPtr;
                }
                return(S_Pos);

        }
        else
         {
                cout<<"\n Invalid input value for new node position";
                return(0);
        }
         }
}
 }

void Doubly_List::Display_List() //Function to display the linked list
{
Node *TempPtrF,*TempPtrB;
 TempPtrF=Dstart;
if(Dstart==NULL)
        cout<<"\n Empty List";
else
{
cout<<"\n Data available in the list are(From first to last):\n";
while(TempPtrF!=NULL)
{
        cout<<TempPtrF->data;
        cout<<"\t";
         TempPtrB=TempPtrF;
        TempPtrF=TempPtrF->next;
}
cout<<"\n Data available in the list are(From last to first):\n";
while(TempPtrB!=NULL)
{
        cout<<TempPtrB->data;
        cout<<"\t";
```

Space for learners:

```
        TempPtrB=TempPtrB->prev;
}
}
}

int main()
{
Doubly_List DL1;
char more='y';
int choice,temp;
clrscr();
while(more=='y' || more=='Y')
{
        cout<<"\n 1. Insert as First Node";
        cout<<"\n 2. Insert as Last Node";
        cout<<"\n 3. Insert at a Specific Position";
        cout<<"\n Enter your choice=";
        cin>>choice;
        switch(choice)
        {
                case 1: DL1.Insert_At_First();
                        cout<<"\n After Insertion::";
                        DL1.Display_List();
                        break;
                case 2: DL1.Insert_At_Last();
                        cout<<"\n After Insertion::";
                        DL1.Display_List();
                        break;
                case 3: temp=DL1.Insert_At_Specific();
                        if(temp==0)
                        cout<<"\nInsertion Unsuccessful";
                        else
                        {
                        cout<<"\n After Insertion::";
                         DL1.Display_List();
                        }
                        break;
                default: cout<<"\n Invalid input for your choice";
         }

        cout<<"\n Input 'y' or 'Y' to insert one more node=";
        cin>>more;
    }
getch();
return 0;
}
```

Space for learners:

2.5.2 Delete Operation on Doubly Linked List

Space for learners:

In this section, the deletion operation on doubly linked list will be discussed and implemented using C++ programming. Let us consider 'Dstart' as the name of the special pointer which stores the memory address of the first node in the list and DeletePtr as the name of the pointer that will be used to point the node which is going to be removed from the Doubly linked list.

Deletion of the first node of a doubly linked list:

To delete the first node, at first, the memory address of the first node that is stored in Dstart has to be assigned to DeletePtr. Then the following steps are performed.

1) The memory address of the second node stored in one address field of the first node is assigned to Dstart so that the second node becomes the first node in the list.
2) The memory address of the earlier first node stored in one address field of the current first node is replaced by NULL.
3) Finally, by using the memory address stored in DeletePtr, the memory allocated for the earlier first node is released.

Figure 2.15 represents the doubly linked list after deleting the first node from the list that is represented in Figure 2.11

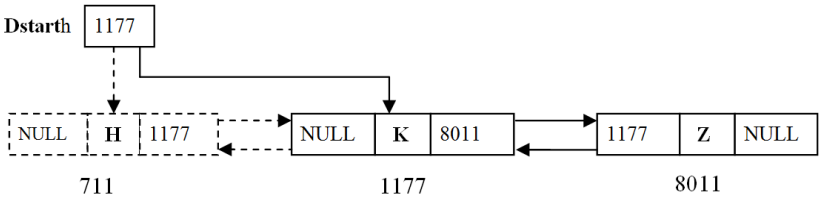


Figure 2.15: Doubly linked list after deletion of the first node

Deletion of the Last node of a doubly linked list:

If both address fields of the first node in a doubly linked list contain NULL then it means that there is only one node available in that list and that node is the first node as well as also the last node. In that case, using the memory address stored in Dstart, the memory

allocated for the last node can be released to delete the last node and then NULL value must be assigned to Dstart.

If the list contains more than one node then to delete the last node, following steps have been performed.

1) At first, previous node to the last node has to be reached from the first node using a pointer. Then this pointer can be used to access the address field of that node so that the memory address of the last node can be accessed.
2) The memory address of the last node is assigned to DeletePtr.
3) The memory address of the last node is replaced by NULL in the address field of the previous node of the last node so that the mentioned previous node becomes the new last node.
4) Using DeletePtr, the memory allocated for the earlier last node is released to delete that node.

Figure 2.16 represents the doubly linked list after deleting the last node from the list that is represented in Figure 2.11.

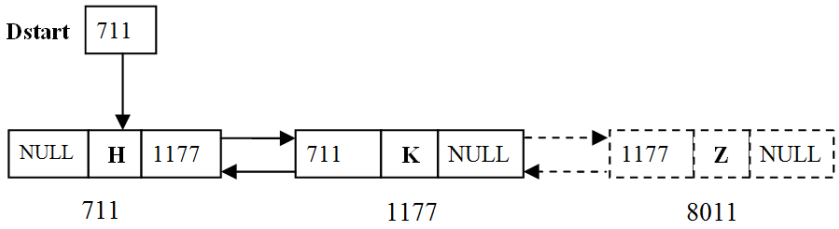


Figure 2.16: Doubly linked list after deletion of the last node

Deletion of a Node Available in a Specific Position in a Doubly linked list:

To delete a node that is available in a specific position in a doubly linked list, at first, the value of the specific position has to be read. Now if the list contains N numbers of nodes then the input value of the specific position cannot be greater than N. In this deletion operation, two pointers are required. Let us consider these as PtrPrev and PtrPos where PtrPos will be used to point the node available at the specific position and PtrPrev will be used to point the previous node to the specific position. Now the following steps have been performed to delete the node available at the specific position and pointed by PtrPos.

Space for learners:

1) The memory address of the next node to the node available at the specific position is assigned to one address field of the node that is pointed by PtrPrev so that the next node to the node pointed by PtrPos becomes new next node to the node that is pointed by PtrPrev.
2) The memory address of the node pointed by PtrPos is replaced by the memory address of the node pointed by PtrPrev in one address field of the node that is the next node to the node pointed by PtrPos. As a result, the node pointed by PtrPrev becomes new previous node to the next node of the node pointed by PtrPos.
3) Finally, using PtrPos, the memory allocated by the node available in the specific position is released to delete that node.

Space for learners:

Figure 2.17 represents the doubly linked list after deleting the 2nd node from the list that is represented in Figure 2.11.

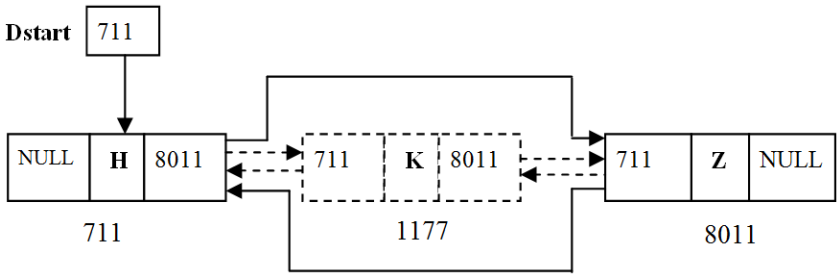


Figure 2.17:Doubly linked list after deletion of the 2nd node

Program 2.4: C++ program to delete existing node from a doubly linked list

```
#include<iostream.h>
#include<conio.h>

struct Node     // User defined data type to create Nodes
{
char data;
struct Node *next;    //Pointer to store memory address of the next
node
struct Node *prev;    //Pointer to store memory address of the
previous node
```

```
};

typedef struct Node Node;

class Doubly_List
{
        private:
                  Node *Dstart;
                int S_Pos,i;
        public:
                  Doubly_List()
                  {
                     Dstart=NULL;
                  }
                void Create_List();
                void Remove_First_Node();
                void Remove_Last_Node();
                int Remove_Specific();
                void Display_List();
};

void Doubly_List::Create_List()    //Function to create a Doubly
linked list
{
        Node *TempPtr;
        char more='y';
        i =1;
        while(more=='y'||more=='Y')
        {
        cout<<"\n Insert "<< I <<"th Node::";
           TempPtr=new Node;
           TempPtr->next=NULL;
           TempPtr->prev=NULL;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(Dstart==NULL)
                 Dstart=TempPtr;
        else
           {
                TempPtr->next=Dstart;
                Dstart->prev=TempPtr;
```

Space for learners:

```
                Dstart=TempPtr;
            }
        i++;
        cout<<"\n Enter 'y' or 'Y' to add one more node=";
        cin>>more;
        }
}
void Doubly_List::Remove_First_Node() // Function to remove the
first node
{
        Node *DeletePtr;

        if(Dstart==NULL)
        cout<<"\n Empty linked list";
        else
        {
        DeletePtr=Dstart;
        if(Dstart->next==NULL)
         {
                Dstart=NULL;
        }
        else
        {
                Dstart=Dstart->next;
                Dstart->prev=NULL;
         }
        delete DeletePtr;
        }
}

void Doubly_List::Remove_Last_Node()  // Function to remove the
last node
{
        Node *DeletePtr,*PtrPrev;

        if(Dstart==NULL)
        {
        cout<<"\n Empty linked list";
        }
        else
        {
```

Space for learners:

```
        DeletePtr=Dstart;
        if(Dstart->next == NULL)
         Dstart= NULL;
        else
        {
                while(DeletePtr->next!=NULL)
                {
                        PtrPrev=DeletePtr;
                        DeletePtr=DeletePtr->next;
                }
                PtrPrev->next=NULL;
        }
        delete DeletePtr;
        }
}

int Doubly_List::Remove_Specific()    /* Function to remove the
node available
at a specific position*/
{
        Node *DeletePtr,*PtrPrev;
        int count=1;

        cout<<"\n Enter the value of the node position=";
        cin>>S_Pos;
        if(Dstart==NULL)
        {
        cout<<"\n Empty linked list";
        return(0);
        }
        else
        {
        DeletePtr=Dstart;
        if(S_Pos==1)
        {
        if(Dstart->next==NULL)
                Dstart=NULL;
        else
        {
                Dstart=Dstart->next;
                Dstart->prev=NULL;
```

Space for learners:

```
        }
        delete DeletePtr;
        return(S_Pos);
        }
        else
        {
        while(DeletePtr->next!=NULL && count<S_Pos)
        {
                count=count+1;
                PtrPrev=DeletePtr;
                 DeletePtr=DeletePtr->next;
        }
        if(count==S_Pos)
         {
                PtrPrev->next=DeletePtr->next;
                if(DeletePtr->next!=NULL)
                 (DeletePtr->next)->prev=PtrPrev;
                delete DeletePtr;
                return(S_Pos);
        }
        else
         {
                cout<<"\n Invalid input value for the node position";
                return(0);
        }
        }
}
 }

void Doubly_List::Display_List() //Function to display the linked list
{
 Node *TempPtrF,*TempPtrB;
 TempPtrF=Dstart;
if(Dstart==NULL)
        cout<<"\n Empty List";
else
{
cout<<"\n Data available in the list are(From first to last):\n";
while(TempPtrF!=NULL)
{
        cout<<TempPtrF->data;
```

Space for learners:

```
        cout<<"\t";
        TempPtrB=TempPtrF;
        TempPtrF=TempPtrF->next;
 }
cout<<"\n Data available in the list are(From last to first):\n";
while(TempPtrB!=NULL)
{
        cout<<TempPtrB->data;
        cout<<"\t";
         TempPtrB=TempPtrB->prev;
}
}
}

int main()
{
Doubly_List DL1;
char more='y';
int choice,temp;
clrscr();
cout<<"\n Create a Doubly Linked List";
 DL1.Create_List();
 DL1.Display_List();
while(more=='y' || more=='Y')
{
        cout<<"\n 1. Delete the First Node";
        cout<<"\n 2. Delete the Last Node";
        cout<<"\n 3. Delete the Node at Specific Position";
        cout<<"\n Enter your choice=";
        cin>>choice;
        switch(choice)
         {
                case 1: DL1.Remove_First_Node();
                        cout<<"\n After Deletion of the First Node::";
                        DL1.Display_List();
                        break;
                case 2: DL1.Remove_Last_Node();
                        cout<<"\n After Deletion of the Last Node::";
                        DL1.Display_List();
                        break;
                case 3: temp=DL1.Remove_Specific();
```

Space for learners:

```
                        if(temp==0)
                        cout<<"\nDeletion Unsuccessful";
                        else
                        {
                        cout<<"\n After Deletion of the Node at
"<<temp<<"th position::";
                        DL1.Display_List();
                        }
                        break;
                default: cout<<"\n Invalid input for your choice";
          }

        cout<<"\n Input 'y' or 'Y' to insert one more node=";
        cin>>more;
}
getch();
return 0;
}
```

Space for learners:

2.5.3 Traversal Operation on Doubly Linked List

In a doubly linked list, the traversal of nodes can be performed sequentially in forward direction (from first node to last node) as well as also in backward direction (from last node to first node). Traversal in both directions is possible because each node in a doubly linked list contains the memory addresses of its next node and previous node.

2.6 CIRCULAR LINKED LIST

Circular linked list is also a linear list of nodes. The structure of a node in a Circular linked list is similar with the structure of the node available in a singly linked list. It means each node in a circular linked list also contains two fields that are a data field and an address field. The data field contains data and the address field contains the memory address of the next node available in the list. The address field of the last node in this type of linked list contains the memory address of the first node. It means, after traversing the last node, we can again traverse the first node in a list. On the other hand, in a singly linked list, it is not possible as the address field of the last node contains 'NULL'. In Figure 2.18, a diagrammatic

representation of a Circular linked is provided where 'Cstart' is a special pointer which stores the memory address of the first node in the list. This special pointer can be used to perform different operations on the list.

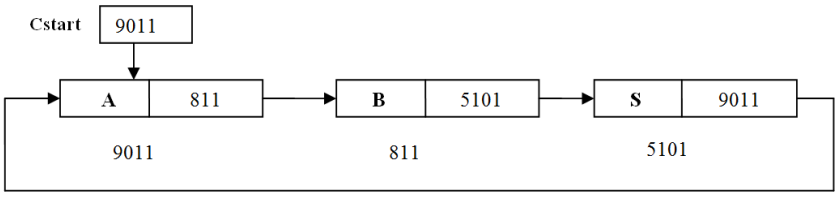


Figure 2.18: Diagrammatic representation of a circular linked list

STOP TO CONSIDER
No NULL link is available in a Circular linked list.

2.6.1 Insert Operation on Circular Linked List

In this section, insertion of a new node to a circular linked list will be discussed and implemented using C++ programming. Let us consider 'Cstart' as the name of the special pointer which stores the memory address of the first node in the list. To insert a new node, at first, a node has to be created by allocating memory at run-time. After memory allocation, the address of the created node is stored in a pointer. Let us consider 'TempPtr' as the name of this pointer. Then appropriate data is assigned to the data field and the memory address stored in TempPtr is assigned to the address field of the newly created node.

Insertion of a new node at the first position in a Circular linked list:

If a circular linked list doesn't contain any node then NULL is stored in Cstart. In that case, to insert a new node, the memory address stored in TempPtr is assigned to Cstart. Otherwise let us try to observe Figure 2.19 to understand the process of inserting a new node at first position in a circular linked list.

Space for learners:

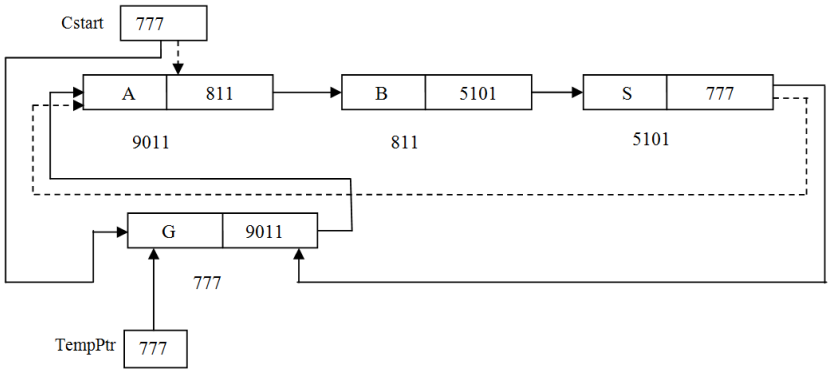


Figure 2.19: Circular linked list after insertion of a new node at the first position

Figure 2.19 represents the circular linked list after inserting a new node at the first position in the Circular linked list that is represented in Figure 2.18. In this insertion operation, following steps have been performed.

1) At first, the last node of the list has to be reached using a pointer. Let us consider PtrLast as the pointer which points to the last node.
2) The memory address of the current first node is assigned to the address field of the new node so that the current first node is linked to the new node as next node.
3) In the third step, the memory address of the new node is assigned to Cstart so that it is linked to the list as the new first node.
4) Finally, the memory address of the new node is assigned to the address field of the node that is pointed by PtrLast so that the new node becomes next node to the last node in the list.

Insertion of a new node at the last position in a Circular linked list:

If a circular linked list doesn't contain any node then the memory address stored in TempPtr is assigned to Cstart. Otherwise let us try to observe Figure 2.20 to understand the process of inserting a new node at the last position in a circular linked list.

Space for learners:

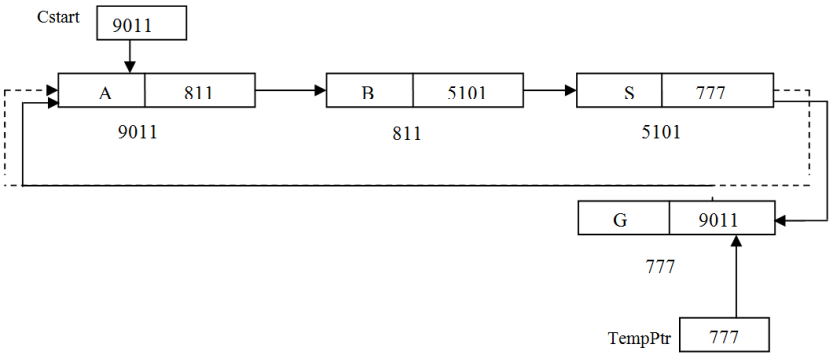


Figure 2.20: Circular linked list after insertion of anew node at the last position

Figure 2.20 represents the circular linked list after inserting a new node at the last position in the Circular linked list that is represented in Figure 2.18. In this insertion operation, following steps have been performed.

1) At first, the last node has to be reached by visiting all the nodes from the first node in the list using a pointer. Let us consider PtrLast as the pointer which points to the last node.
2) In the second step, the memory address of the new node is assigned to the address field of the current last node pointed by PtrLast. As a result, the new node is linked as next node to the current last node.
3) Finally, the memory address of the first node is assigned to the address field of the new node.

Insertion of a new node at a specific position in a circular linked list:

To insert a new node at a specific position, at first the specific position has to be read. Let us try to observe Figure 2.21 to understand the process of inserting a new node at a specific position that is 3 in the mentioned figure.

Space for learners:

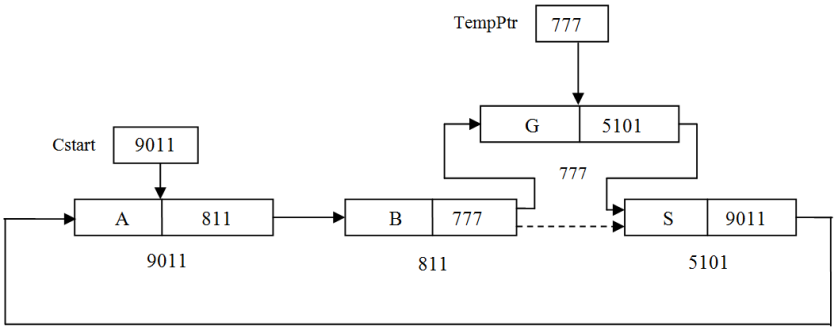


Figure 2.21: Circular linked list after insertion of a new node at the 3rdposition

Figure 2.21 represents the circular linked list after inserting a new node at 3rd position in the circular linked list that is represented in figure 2.18. In this process of insertion operation, following steps have been performed.

1) At first, the second node has to be reached using a pointer.
2) In the second step, the memory address stored in the address field of the second node is assigned to the address field of the newly created node.
3) Finally, the memory address of the new node is assigned to the address field of the second node. As a result, the new node is inserted at the 3rd position in the list.

Program 2.5: C++ program to insert new node to a circular linked list

```
#include<iostream.h>
#include<conio.h>

struct Node     // User defined data type to create Nodes
{
        char data;
        struct Node *next; // Pointer to point the next node
};

typedef struct Node Node;

class Circular_List
```

Space for learners:

```
{
        private:
        Node *Cstart;
        int S_Pos;
        public:
         Circular_List()
        {
        Cstart=NULL;
        }
        void Insert_At_First();
        void Insert_At_Last();
        int Insert_At_Specific();
        void Display_List();
};

void Circular_List::Insert_At_First() // Function to insert node at
        first position
{
        Node *TempPtr,*PtrLast;
        TempPtr= new Node;
        TempPtr->next= TempPtr;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(Cstart==NULL)
          Cstart=TempPtr;
        else
        {
           PtrLast=Cstart;
        while(PtrLast->next!=Cstart)
        PtrLast=PtrLast->next;
           TempPtr->next=Cstart;
           PtrLast->next=TempPtr;
           Cstart=TempPtr;
        }
}

void Circular_List::Insert_At_Last()  // Function to insert node at
        Last position
{
        Node *TempPtr,*PtrLast;
        TempPtr= new Node;
```

Space for learners:

```
        TempPtr->next= TempPtr;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(Cstart==NULL)
        Cstart=TempPtr;
        else
        {
        PtrLast=Cstart;
        while(PtrLast->next!=Cstart)
        PtrLast=PtrLast->next;
        PtrLast->next=TempPtr;
        TempPtr->next=Cstart;
        }
}


int Circular_List::Insert_At_Specific()   // Function to insert node at
        a specific position
{
        Node *TempPtr,*PtrPrev,*PtrLast;
        int count=1;
        TempPtr= new Node;
        TempPtr->next= TempPtr;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        cout<<"\n Enter the value for new node position=";
        cin>> S_Pos;
        if(Cstart==NULL)
        {
        if(S_Pos==1)
         {
        Cstart=TempPtr;
        return(S_Pos);
        }
        else
        {
        cout<<"\n Invalid input value for new node position";
        return(0);
        }
        }
        else
```

Space for learners:

```
        {
        if(S_Pos==1)
          {
        PtrLast=Cstart;
        while(PtrLast->next!=Cstart)
        PtrLast=PtrLast->next;
        TempPtr->next=Cstart;
        PtrLast->next=TempPtr;
         Cstart=TempPtr;
        return(S_Pos);
          }
        else
          {
        PtrPrev=Cstart;
        while(PtrPrev->next!=Cstart && count<S_Pos-1)
        {
        count=count+1;
          PtrPrev=PtrPrev->next;
        }
        if(count==S_Pos-1)
        {
         TempPtr->next=PtrPrev->next;
         PtrPrev->next=TempPtr;
        return(S_Pos);
        }
        else
        {
        cout<<"\n Invalid input value for new node position";
        return(0);
        }
          }
}
}

void Circular_List::Display_List() //Function to display the linked
        list
{
Node *TempPtr;
TempPtr=Cstart;
if(Cstart==NULL)
        cout<<"\n Empty List";
```

Space for learners:

```
else
{
cout<<"\n Data available in the list are:\n";
while(TempPtr->next!=Cstart)
{
        cout<<TempPtr->data;
        cout<<"\t";
        TempPtr=TempPtr->next;
}
cout<<TempPtr->data;
}
}

int main()
{
Circular_List CL1;
char more='y';
int choice,temp;
clrscr();
while(more=='y' || more=='Y')
   {
        cout<<"\n 1. Insert as First Node";
        cout<<"\n 2. Insert as Last Node";
        cout<<"\n 3. Insert at a Specific Position";
        cout<<"\n Enter your choice=";
        cin>>choice;
        switch(choice)
         {
        case 1: CL1.Insert_At_First();
                cout<<"\n After Insertion::";
                CL1.Display_List();
                break;
        case 2: CL1.Insert_At_Last();
                cout<<"\n After Insertion::";
                CL1.Display_List();
                break;
        case 3: temp=CL1.Insert_At_Specific();
                if(temp==0)
                cout<<"\nInsertion Unsuccessful";
                else
                {
```

Space for learners:

```
                cout<<"\n After Insertion::";
                CL1.Display_List();
                }
                break;
        default: cout<<"\n Invalid input for your choice";
          }

        cout<<"\n Input 'y' or 'Y' to insert one more node=";
        cin>>more;
    }
getch();
return 0;
}
```

Space for learners:

2.6.2 Delete Operation on Circular Linked List

In this section, the deletion operation on circular linked list will be discussed and implemented using C++ programming. Let us consider 'Cstart' as the name of the special pointer which stores the memory address of the first node and DeletePtr as the name of the pointer that will be used to point the node which is going to be removed from the list.

.

Deletion of the first node of a circular linked list:

To delete the first node, at first, the memory address of the first node is assigned to DeletePtr. If the address field of the first node is pointing itself then it means there is only one node available in the list. In that case, using DeletePtr, the memory allocated by the first node is released to delete that node. Then NULL value is assigned to Cstart. In case of a list with more than one node, following steps have been performed to delete the first node.

1) Last node has to be reached using a pointer. Let us consider PtrLast as the pointer which points to the last node.
2) The memory address stored in the address field of the first node is assigned to Cstart so that the second node becomes first node in the list.
3) The memory address of the new first node is assigned to the address field of the last node that is pointed by PtrLast.

4) Finally, by using DeletePtr, the memory allocated for the earlier first node is released.

Figure 2.22 represents the Circular linked list after deleting the first node from the list that is represented in Figure 2.18.

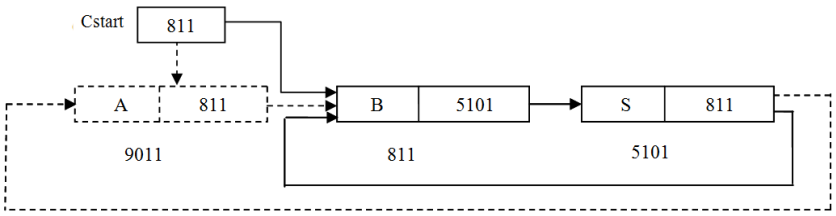


Figure 2.22: Circular linked list after deletion of the first node

Deletion of the last node of a circular linked list:

If the address field of the first node is pointing itself then it means there is only one node available in the Circular linked list. In that case, using Cstart, the memory allocated by the first or the last node is released to delete that node. Then NULL value is assigned to Cstart.

If the list contains more than one node then to delete the last node, following steps have been performed.

Previous node to the last node has to be reached from the first node using a pointer.

Let us consider PtrPrevLast as the pointer which points the previous node of the last node.

The memory address of the last node is assigned to DeletePtr.

The memory address of the first node is assigned to the address field of the node that is pointed by PtrPrevLast so that it becomes the new last node in the list.

Finally, using DeletePtr, the memory allocated for the earlier last node is released to delete that node.

Figure 2.23 represents the circular linked list after deleting the last node from the list that is represented in Figure 2.18.

Space for learners:

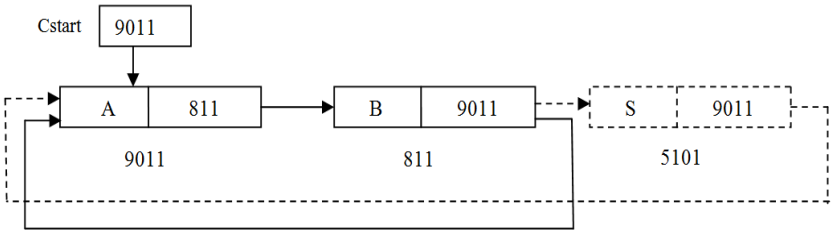


Figure 2.23: Circular linked list after deletion ofthe last node

Deletion of a node available in a specific position in a circular linked list:

To delete a node that is available in a specific position in a Circular linked list, at first, the value of the specific position has to be read. In this deletion operation, two pointers are required. Let us consider these as PtrPrev and PtrPos where PtrPos will be used to point the node available at the specific position and PtrPrev will be used to point its previous node. Then the following steps have been performed to delete the specific node.

1) The memory address of the next node to the node pointed by PtrPos is assigned to the address field of the node that is pointed by PtrPrev.
2) Finally, using PtrPos, the memory allocated by the node available in the specific position is released.

Figure 2.24 represents the Circular linked list after deleting the 2nd node from the list that is represented in Figure 2.18.

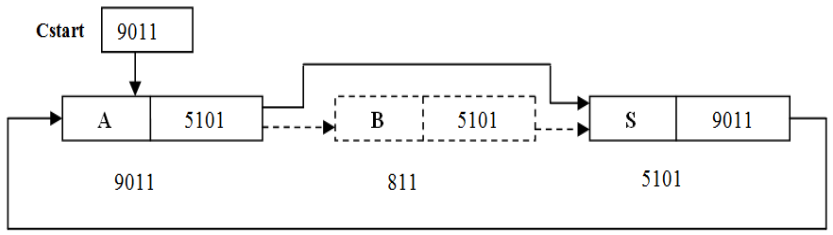


Figure 2.24: Circular linked list after deletion of the 2nd node

Space for learners:

2.6.3 Traversal Operation on Circular Linked List

Space for learners:

We have already learnt that each node in a Circular linked list stores the memory address of its next node and the last node contains the memory address of the first node. So, traversal operation can be performed from the first node to the last node by accessing the memory address stored in each node. The memory address of the first node can be obtained from the special pointer associated with the list. The traversal operation can be stopped when the node is reached which contains the address of the first node that means when the last node in the list is reached. But if required, traversal can be continued after reaching the last node.

STOP TO CONSIDER

Traversal from one node to any other node is possible in Circular linked list.

Program 2.6: C++program to delete existing node from a Circular linked list.

```
#include<iostream.h>
#include<conio.h>

struct Node     // User defined data type to create Nodes
{
        char data;
        struct Node *next; //Pointer to point the next node
};

typedef struct Node Node;

class Circular_List
{
        private:
                Node *Cstart;
               int S_Pos,i;
        public:
                Circular_List()
                {
                   Cstart=NULL;
                }
```

```
                void Create_List();
                void Remove_First_Node();
                void Remove_Last_Node();
                int Remove_Specific();
                void Display_List();
};

void Circular_List::Create_List()  // Function to create a Circular linked list
{
        Node *TempPtr,*PtrLast;
        char more='y';
        i =1;
        while(more=='y'||more=='Y')
        {
        cout<<"\n Insert "<< i <<"th Node::";
           TempPtr=new Node;
           TempPtr->next=TempPtr;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(Cstart==NULL)
                 Cstart=TempPtr;
        else
           {
                PtrLast=Cstart;
                while(PtrLast->next!=Cstart)
                    PtrLast=PtrLast->next;
                PtrLast->next=TempPtr;
                TempPtr->next=Cstart;
           }
        i++;
        cout<<"\n Enter 'y' or 'Y' to add one more node=";
        cin>>more;
        }
}

void Circular_List::Remove_First_Node() // Function to remove the first node
{
        Node *DeletePtr,*PtrLast;
```

Space for learners:

```
        if(Cstart==NULL)
        cout<<"\n Empty linked list";
        else
        {
        DeletePtr=Cstart;
        if(Cstart==Cstart->next)
                Cstart=NULL;
        else
         {
                PtrLast=Cstart;
                while(PtrLast->next!=Cstart)
                PtrLast=PtrLast->next;
                Cstart=Cstart->next;
                PtrLast->next=Cstart;
           }
        delete DeletePtr;
        }
}

void Circular_List::Remove_Last_Node()  // Function to remove the
last node
{
        Node *DeletePtr,*PtrPrev;

        if(Cstart==NULL)
        {
        cout<<"\n Empty linked list";
        }
        else
        {
        if(Cstart==Cstart->next)
         {
                DeletePtr=Cstart;
                Cstart=NULL;
        }
        else
        {
                DeletePtr=Cstart;
                while(DeletePtr->next!=Cstart)
                {
                    PtrPrev=DeletePtr;
```

Space for learners:

```
                DeletePtr=DeletePtr->next;
            }
            PtrPrev->next=Cstart;
          }
        delete DeletePtr;
        }
}

int Circular_List::Remove_Specific()    /* Function to remove the node available
at a specific position*/
{
        Node *DeletePtr,*PtrPrev,*PtrLast;
        int count=1;

        cout<<"\n Enter the value of the node position=";
        cin>>S_Pos;
        if(Cstart==NULL)
        {
        cout<<"\n Empty linked list";
        return(0);
        }
        else
        {

        if(S_Pos==1)
        {
         DeletePtr=Cstart;
        if(Cstart==Cstart->next)
                Cstart=NULL;

        else
         {
                 PtrLast=Cstart;
                while(PtrLast->next!=Cstart)
                PtrLast=PtrLast->next;
                Cstart=Cstart->next;
                 PtrLast->next=Cstart;
                }
                delete DeletePtr;
                return(S_Pos);
```

Space for learners:

```
        }
        else
        {
        DeletePtr=Cstart;
        while(DeletePtr->next!=Cstart && count<S_Pos)
         {
                count=count+1;
                PtrPrev=DeletePtr;
                DeletePtr=DeletePtr->next;
        }
        if(count==S_Pos)
        {
                PtrPrev->next=DeletePtr->next;
                delete DeletePtr;
                return(S_Pos);
        }
        else
        {
                cout<<"\n Invalid input value for the node position";
                return(0);
        }
        }
}
 }

void Circular_List::Display_List() //Function to display the Circular
linked list
{
   Node *TempPtr;
   TempPtr=Cstart;
if(Cstart==NULL)
        cout<<"\n Empty List";
else
   {
cout<<"\n Data available in the list are:\n";
while(TempPtr->next!=Cstart)
{
        cout<<TempPtr->data;
        cout<<"\t";
        TempPtr=TempPtr->next;
}
```

Space for learners:

```
cout<<TempPtr->data;
}
}

int main()
{
   Circular_List CL1;
char more='y';
int choice,temp;
clrscr();
cout<<"\n Create a Singly Linked List";
   CL1.Create_List();
   CL1.Display_List();
while(more=='y' || more=='Y')
   {
        cout<<"\n 1. Delete the First Node";
        cout<<"\n 2. Delete the Last Node";
        cout<<"\n 3. Delete the Node at Specific Position";
        cout<<"\n Enter your choice=";
        cin>>choice;
        switch(choice)
         {
                case 1: CL1.Remove_First_Node();
                        cout<<"\n After Deletion of the First Node::";
                        CL1.Display_List();
                        break;
                case 2: CL1.Remove_Last_Node();
                        cout<<"\n After Deletion of the Last Node::";
                        CL1.Display_List();
                        break;
                case 3: temp=CL1.Remove_Specific();
                        if(temp==0)
                        cout<<"\nDeletion Unsuccessful";
                        else
                        {
                        cout<<"\n After Deletion of the Node at
"<<temp<<"th position::";
                           CL1.Display_List();
                        }
                        break;
                default: cout<<"\n Invalid input for your choice";
```

Space for learners:

```
        }

        cout<<"\n Input 'y' or 'Y' to insert one more node=";
        cin>>more;
    }
getch();
return 0;
}
```

Space for learners:

2.7 DOUBLY CIRCULAR LINKED LIST

A new type of linked list can be developed by combining the concept of doubly linked list and Circular linked list. This new type of linked list is termed as doubly circular linked list where the structure of each node is similar to the structure of the node in a doubly linked list. In this linked list, one address field of the first node contains the memory address of the last node in the list. Again, one address field of the last node contains the memory address of the first node in the list. A diagrammatic representation of this type of linked list is provided in Figure 2.25 where 'DCstart' is a special pointer which stores the memory address of the first node in the list.This special pointer can be used to perform different operations on the linked list.Figures from Figure 2.26 to Figure 2.30 represent the insertion and deletion operations onthedoubly circular linked list that is presented in Figure 2.25. These operations in doubly circular linked list are implemented in Program 2.7.

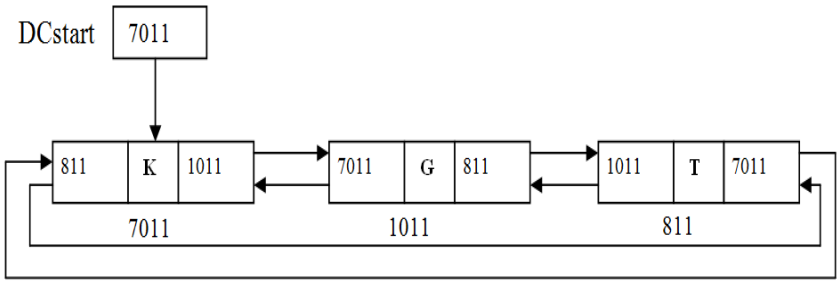


Figure 2.25: Diagrammatic representation of a doubly circular linked list.

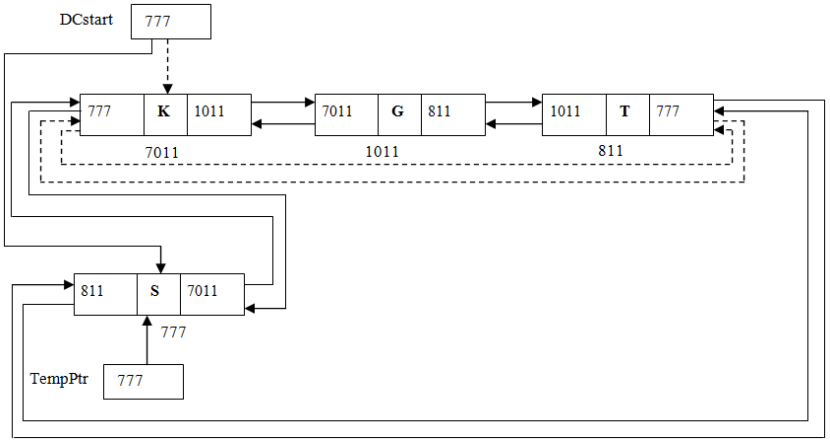


Figure 2.26: Doubly circular linked list after insertion of a node at first position

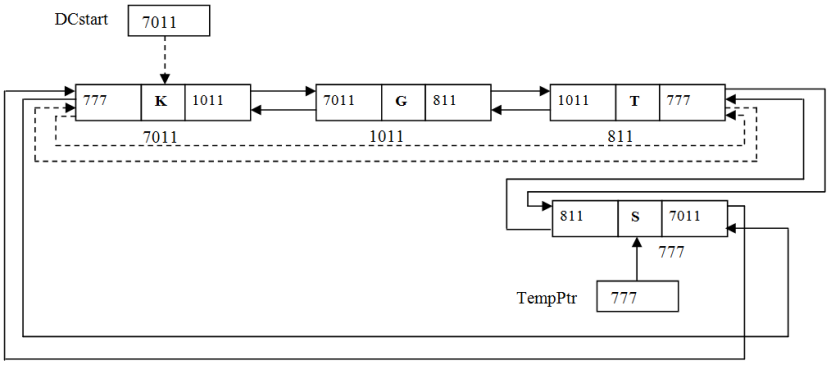


Figure 2.27: Doubly circular linked list after insertion of a node at last position

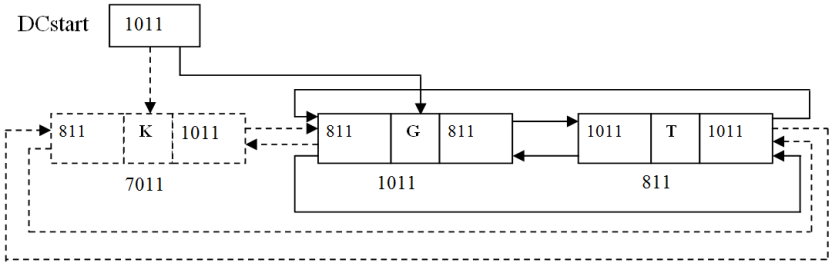


Figure 2.28: Doubly circular linked list after deletion of the first node

Space for learners:

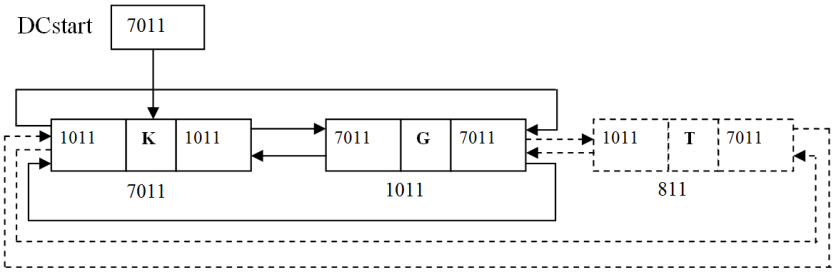


Figure 2.29: Doubly circular linked list after deletion of the last node

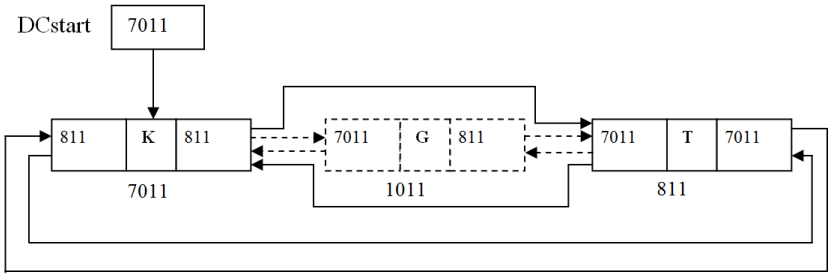


Figure 2.30: Doubly circular linked list after deletion of the node available at 2nd position

Program 2.7: C++ program to implement doubly circular linked list

```
#include<iostream.h>
#include<conio.h>

struct Node     // User defined data type to create Nodes
{
        char data;
        struct Node *next; //Pointer to point the next node
        struct Node *prev; //Pointer to point the previous node
};

typedef struct Node Node;

class DoublyCircular
{
        private:
                Node *DCstart,*DClast;
              int S_Pos,i;
        public:
              DoublyCircular()
```

Space for learners:

```
                {
                   DCstart=NULL;
                }
               void Create_List();
               void Insert_At_First();
               void Insert_At_Last();
               int Insert_At_Specific();
               void Remove_First_Node();
               void Remove_Last_Node();
               int Remove_Specific();
               void Display_List();
};

void DoublyCircular::Create_List()    //Function to create a Doubly
circular linked list
{
        Node *TempPtr;
        char more='y';
        i =1;
        while(more=='y'||more=='Y')
        {
        cout<<"\n Insert "<< i <<"th Node::";
           TempPtr=new Node;
           TempPtr->next=TempPtr;
           TempPtr->prev=TempPtr;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(DCstart==NULL)
           {
                DCstart=TempPtr;
                DClast=TempPtr;
           }
        else
           {
                TempPtr->next=DCstart;
                DCstart->prev=TempPtr;
                DCstart=TempPtr;
                DCstart->prev=DClast;
                DClast->next=DCstart;
           }
        i++;
```

Space for learners:

```
        cout<<"\n Enter 'y' or 'Y' to add one more node=";
        cin>>more;
        }
}

void DoublyCircular::Insert_At_First() // Function to insert node at
first position
{
        Node *TempPtr;
        TempPtr= new Node;
        TempPtr->next= TempPtr;
        TempPtr->prev= TempPtr;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(DCstart==NULL)
{
        DCstart=TempPtr;
DClast=TempPtr;
}
        else
        {
        TempPtr->next=DCstart;
        DCstart->prev=TempPtr;
        DCstart=TempPtr;
DCstart->prev=DClast;
 DClast->next=DCstart;
        }
}

void DoublyCircular::Insert_At_Last()  // Function to insert node at
Last position
{
        Node *TempPtr;
        TempPtr= new Node;
        TempPtr->next= TempPtr;
        TempPtr->prev= TempPtr;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        if(DCstart==NULL)
{
         DCstart=TempPtr;
```

Space for learners:

```
DClast=TempPtr;
}
        else
        {
        TempPtr->prev=DClast;
TempPtr->next=DCstart;
        DClast->next=TempPtr;
        DCstart->prev=TempPtr;
        }
}

int DoublyCircular::Insert_At_Specific()   // Function to insert node
at a specific position
{
        Node *TempPtr,*PtrPrev;
        int count=1;
        TempPtr= new Node;
        TempPtr->next= TempPtr;
        TempPtr->prev= TempPtr;
        cout<<"\n Enter a character=";
        cin>>TempPtr->data;
        cout<<"\n Enter the value for new node position=";
        cin>>S_Pos;
        if(DCstart==NULL)
        {
        if(S_Pos==1)
        {
        DCstart=TempPtr;
DClast=TempPtr;
        return(S_Pos);
        }
        else
        {
        cout<<"\n Invalid input value for new node position";
        return(0);
        }
        }
        else
        {
        if(S_Pos==1)
        {
```

Space for learners:

```
        TempPtr->next=DCstart;
        DCstart->prev=TempPtr;
        DCstart=TempPtr;
DCstart->prev=DClast;
DClast->next=DCstart;
        return(S_Pos);
        }
        else
        {
         PtrPrev=DCstart;
        while(PtrPrev->next!=DCstart && count<S_Pos-1)
        {
                count=count+1;
                 PtrPrev=PtrPrev->next;
         }
        if(count==S_Pos-1)
         {
                if(PtrPrev->next==DCstart)
                {
                 PtrPrev->next=TempPtr;
                TempPtr->prev=PtrPrev;
DClast=TempPtr;
DClast->next= DCstart;
 DCstart->prev=DClast;
                }
                else
                {
                        TempPtr->next=PtrPrev->next;
                        TempPtr->prev=PtrPrev;
                        (PtrPrev->next)->prev=TempPtr;
                        PtrPrev->next=TempPtr;
                 }
                return(S_Pos);

        }
        else
          {
                cout<<"\n Invalid input value for new node position";
                return(0);
        }
         }
```

Space for learners:

```
 }
}


void DoublyCircular::Remove_First_Node() // Function to remove
the first node
{
        Node *DeletePtr;

        if(DCstart==NULL)
        cout<<"\n Empty linked list";
        else
        {
        DeletePtr=DCstart;
        if(DCstart->next==DCstart)
        {
                DCstart=NULL;
DClast=NULL;
          }
        else
          {
                DCstart=DCstart->next;
                DCstart->prev=DClast;
DClast->next=DCstart;
          }
        delete DeletePtr;
        }
}

void DoublyCircular::Remove_Last_Node()  // Function to remove
the last node
{
        Node *DeletePtr,*PtrPrev;

        if(DCstart==NULL)
        {
        cout<<"\n Empty linked list";
        }
        else
        {
```

Space for learners:

```
        if(DCstart->next == DCstart)
 {
DeletePtr=DCstart;
        DCstart= NULL;
DClast= NULL;
}

        else
        {
DeletePtr=DClast;
                PtrPrev=DClast->prev;
 DClast=PtrPrev;
                DClast->next=DCstart;
DCstart->prev=DClast;
          }
        delete DeletePtr;
}
}

int DoublyCircular::Remove_Specific()   /* Function to remove the
node available
at a specific position*/
{
        Node *DeletePtr,*PtrPrev;
        int count=1;

        cout<<"\n Enter the value of the node position=";
        cin>>S_Pos;
        if(DCstart==NULL)
        {
        cout<<"\n Empty linked list";
        return(0);
        }
        else
        {

        if(S_Pos==1)
        {
DeletePtr=DCstart;
        if(DCstart->next==DCstart)
{
```

Space for learners:

```
                DCstart=NULL;
 DClast= NULL;
 }
        else
        {
                DCstart=DCstart->next;
                DCstart->prev=DClast;
DClast->next=DCstart;
         }
        delete DeletePtr;
        return(S_Pos);
         }
        else
        {
        DeletePtr=DCstart;
        while(DeletePtr->next!=DCstart && count<S_Pos)
         {
                count=count+1;
                PtrPrev=DeletePtr;
                 DeletePtr=DeletePtr->next;
        }
        if(count==S_Pos)
         {

                if(DeletePtr->next==DCstart)
{
DClast = PtrPrev;
DClast->next=DCstart;
DCstart->prev=DClast;
}
else
 {
PtrPrev->next=DeletePtr->next;
                  (DeletePtr->next)->prev=PtrPrev;
}
                delete DeletePtr;
                return(S_Pos);
         }
        else
         {
                cout<<"\n Invalid input value for the node position";
```

Space for learners:

```
                return(0);
            }
              }
      }
}


void DoublyCircular::Display_List() //Function to display the linked
list
{
Node *TempPtr;
TempPtr=DCstart;
if(DCstart==NULL)
        cout<<"\n Empty List";
else
{
cout<<"\n Data available in the list are(From first to last):\n";
while(TempPtr->next!=DCstart)
  {
        cout<<TempPtr->data;
        cout<<"\t";
        TempPtr=TempPtr->next;
}
cout<<TempPtr->data;

cout<<"\n Data available in the list are(From last to first):\n";
TempPtr=DClast;
while(TempPtr->prev!=DClast)
{
        cout<<TempPtr->data;
        cout<<"\t";
        TempPtr=TempPtr->prev;
}
cout<<TempPtr->data;

 }
}

int main()
{
 DoublyCircular DCL1;
```

Space for learners:

```
char more='y';
int choice,temp;
clrscr();
cout<<"\n Create a Doubly Circular Linked List";
DCL1.Create_List();
DCL1.Display_List();
while(more=='y' || more=='Y')
{
cout<<"\n 1. Insert as First Node";
        cout<<"\n 2. Insert as Last Node";
        cout<<"\n 3. Insert at a Specific Position";
        cout<<"\n 4. Delete the First Node";
        cout<<"\n 5. Delete the Last Node";
        cout<<"\n 6. Delete the Node at Specific Position";
        cout<<"\n Enter your choice=";
        cin>>choice;
        switch(choice)
         {
case 1: DCL1.Insert_At_First();
                        cout<<"\n After Insertion::";
                        DCL1.Display_List();
                        break;
                case 2: DCL1.Insert_At_Last();
                        cout<<"\n After Insertion::";
                        DCL1.Display_List();
                        break;
                case 3: temp=DCL1.Insert_At_Specific();
                        if(temp==0)
                        cout<<"\nInsertion Unsuccessful";
                        else
                        {
                        cout<<"\n After Insertion::";
                        DCL1.Display_List();
                        }
                        break;
                case 4: DCL1.Remove_First_Node();
                        cout<<"\n After Deletion of the First Node::";
                        DCL1.Display_List();
                        break;
                case 5: DCL1.Remove_Last_Node();
                        cout<<"\n After Deletion of the Last Node::";
```

Space for learners:

```
                    DCL1.Display_List();
                    break;
            case 6: temp=DCL1.Remove_Specific();
                    if(temp==0)
                    cout<<"\nDeletion Unsuccessful";
                    else
                    {
                    cout<<"\n After Deletion of the Node at
"<<temp<<"th position::";
                       DCL1.Display_List();
                    }
                    break;
            default: cout<<"\n Invalid input for your choice";
          }

        cout<<"\n Input 'y' or 'Y' to insert one more node=";
        cin>>more;
    }
getch();
return 0;
}
```

CHECK YOUR PROGRESS

1. Multiple choice question:

A. Which of the following linked list does not contain any node with NULL value in its address field?

(i) Singly linked list
(ii) Doubly linked list
(iii) Circular linked list
(iv) All of the above

B. Which of the following linked list allocates more memory than other linked lists?

(i) Singly linked list
(ii) Doubly linked list
(iii) Circular linked list
(iv) All of the above

Space for learners:

C. Which of the following is not true in case of linked list?
   (i) Data are stored in contiguous memory locations.
   (ii) Each node consists of data field and address fields.
   (iii) Direct access of data is not possible.
   (iv) None of these

D. Traversal from a node to any other node is possible in _____.
   (i) Singly linked list
   (ii) Circular linked list
   (iii) Doubly linked list
   (iv) Both (ii) and (iii)

E. At most _____ pointers are modified to delete a node from a doubly linked list.
   (i) One
   (ii) Two
   (iii) Three
   (iv) Four

2. State whether the following statements are true or false:

   A. The last node of a circular linked list contains the memory address of the first node.
   B. Traversal from last node to first node is not possible in doubly linked list.
   C. Data in a linked list can be accessed randomly.
   D. When the first node in a circular linked list contains the memory address of itself then it means that the list contains only one node.

Space for learners:

2.8 SUMMING UP

Linked list is a dynamic data structure. A linked list is a collection of nodes where each node contains one data field and one or more address fields. Data field contains data and address fields stores memory address of nodes. In linked list, data may not be stored in contiguous memory locations like arrays.

Three basic types of linked list are singly linked list, doubly linked list and circular linked list.

In singly linked list, each node contains one data field and one address field. Data field contains data and address field contains the address of the next node. The address field of the last node contains NULL value as there is no next node available.

In doubly linked list, each node contains one data field and two address fields. One address field contains the memory address of the previous node and the second address field contains the memory address of the next node. One address field of the first node contains NULL value as there is no previous node available. Similarly, one address field of the last node contains NULL value as there is no next node available.

In circular linked list, each node contains one data field and one address field like singly linked list. It is similar to singly linked list but the address field of the last node in a circular linked list stores the memory address of the first node.

Doubly circular linked list can be developed by combining the concepts of doubly linked list and circular linked list.

Space for learners:

2.9 ANSWERS TO CHECK YOUR PROGRESS

1. A. (iii) B. (ii) C. (i) D.(iv) E.(ii)
2. A. True B. False C. False D. True

2.10 POSSIBLE QUESTIONS

1) Write down the advantages and disadvantages of linked list over array.
2) Write down any four applications of linked list.
3) Write down a C++ program to delete the previous node to the 5^{th} node in a circular linked list.
4) Write down a C++ program to insert a new node after 3^{rd} node in a doubly linked list.
5) Write down a C++ program to reverse a singly linked list.
6) How can we say that singly linked list is better than doubly linked list?

2.11 REFERENCES AND SUGGESTED READINGS

1) Seymour Lipschutz : Data Structures With C, Tata McGraw-Hill
2) Ellis Horowitz, Sartaj Sahni : Fundamentals of data structures, Computer Science Press
3) Yedidyah Langsam,Moshe J. Augenstein, Aaron M.Tenenbaum: Data structures using C and C++ , Prentice-Hall India

---×---

Space for learners:

UNIT 3: STACK AND QUEUE

Unit Structure:

Space for learners:

3.1 INTRODUCTION

Stack and queue are two very important data structures. Both of these are linear data structures. These can be implemented using both array and linked list. One important point regarding these structures is that insertion and deletion operations can be performed only at the beginning or ending position. In this unit, we are going to learn about these data structures and implement both using C++ programming.

3.2 UNIT OBJECTIVES

After reading this unit you are expected to be able to learn:

- Definition of stack
- About applications of stack
- Implementations of stack using C++ programming
- Definition of queue

- Applications of queue
- Implementations of queue using C++ programming
- About circular queue and its implementation
- About Priority queue and its implementation

Space for learners:

3.3 STACK

Stack can be defined as a linear data structure where the last data inserted will be removed first. It is a one ended structure where both insertion and deletion operations are performed at the same end. This end can be termed as Top end. Insertion of a new data to a stack is termed as Push operation and deletion of an existing data from a stack is called as Pop operation. Stack can also be termed as Last-In First-Out (LIFO) data structure.

STOP TO CONSIDER

Stack can also be termed as First In Last Out structure

3.3.1 Applications of Stack

Stack is used in various important applications. A few of these are mentioned below.

1) Stack is used to convert an infix expression to its postfix expression
2) Stack is used to convert an infix expression to its prefix expression
3) Stack is used in reversing strings
4) Stack is used to implement Depth First Search (DFS) traversal algorithm to traverse graph.
5) Stack is used to process function calls. When a function is called from another function then stack is used to store the information about the calling function like memory address of the calling function.
6) Stack is also used in operating system for memory management.

3.3.2 Implementation of Stack Using Array

Space for learners:

In this section, the implementation of stack using array is going to be discussed. In this implementation, at first, an array has to be declared to store data. Let us consider Stack_Arr as the array and the size of this array is Stack_Size. It means the stack can be able to store at most Stack_Size numbers of data. Then one integer variable is also required for this implementation. Let us consider Top as the name of this variable. Initially Top is assigned with -1 which means that the stack is empty.

To perform Push operation on a stack, at first, overflow condition must be checked. If the overflow condition becomes true then it means that the stack doesn't have any empty space to insert a new data. The overflow condition using C++ statement is presented below.

```
if ( Top = = Stack_Size - 1 )
```

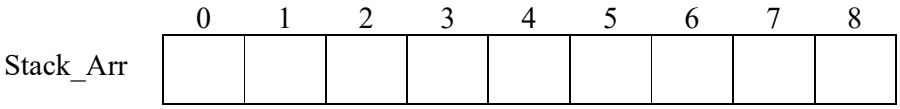
If overflow condition is not true then new data can be inserted in the empty cell that is next to the cell pointed by Top. For this purpose at first Top is incremented by one and then using the array name and the incremented Top, new data can be inserted or pushed to the appropriate cell in the stack.

To perform Pop operation on a stack, at first underflow condition must be checked. If underflow condition is true then it means that the stack is empty. The underflow condition using C++ statement is presented below.

```
if ( Top = = -1)
```

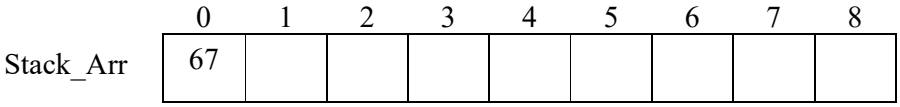
If underflow condition is not true then the data pointed by Top is considered to be deleted from the stack and Top is decremented by one. As a result, Top will point to the cell which stores the next last inserted data. If the value of Top is 0 then it means that there is only one data available in the stack. In this situation, after pop operation, Top will contain -1.

Let us try to observe the following diagrammatic presentations of a stack after Push and Pop operations so that array implementation of stack can be visualized.



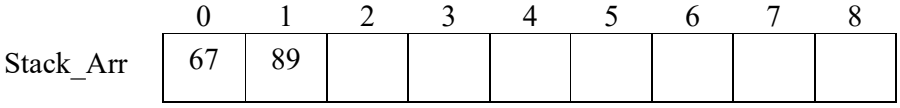
Top = -1

Figure 3.1(a): Empty Stack



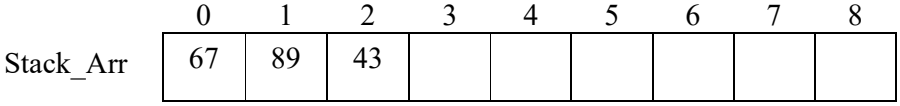
Top = 0

Figure 3.1 (b): StackAfter First Push Operation



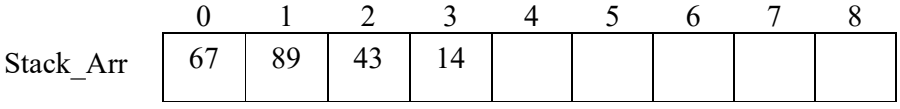
Top = 1

Figure 3.1 (c): StackAfter Second Push Operation



Top =2

Figure 3.1 (d): StackAfter Third Push Operation



Top = 3

Figure 3.1 (e): StackAfter Fourth Push Operation

Space for learners:

	0	1	2	3	4	5	6	7	8
Stack_Arr	67	89	43	14	7				

Top =4

Figure 3.1 (f): StackAfter Fifth Push Operation

	0	1	2	3	4	5	6	7	8
Stack_Arr	67	89	43	14					

Top = 3

Figure 3.1 (g): StackAfter First Pop Operation

	0	1	2	3	4	5	6	7	8
Stack_Arr	67	89	43						

Top =2

Figure 3.1 (h): StackAfter Second Pop Operation

Program 3.1: C++ program to implement stack using array

```
# include <iostream.h>
# include <conio.h>
# define Stack_Size 50

class Stack    // Class to implement stack using array
{
private:
   intStack_Arr[Stack_Size];       // Array to store stack data
   int Top, Data;
public:
   Stack()
```

Space for learners:

```
    {
        Top = -1;
    }
    int Push(int);
    int Pop();
    voidDisplay_Stack();
};

int Stack :: Push(int data)            // Push Operation
{
if( Top == Stack_Size -1 )      //Overflow condition
 {
    cout<<"\n Stack Overflow";
    return(0);
}
else
{
    Top = Top + 1;
    Stack_Arr[Top] = data;
    return(1);
 }
}

int Stack :: Pop()        // Pop Operation
{
int Deleted;
if( Top == -1)// Underflow condition
{
    cou<<"\n Stack Underflow";
    return -99;
}
else
{
    Deleted = Stack_Arr[Top];
    Top = Top - 1;
    return Deleted;
}
}
```

Space for learners:

```
void Stack :: Display_Stack()    //  Function to display data available
in the stack
{
int i;
if(Top == -1)            // Underflow condition
cout<<"\n Stack Underflow";
else
{
cout<<"\n The Stack elements are::\n";
for( i =0; i <=Top ; i++)
    cout<<"\t"<<Stack_Arr[i];
}
}

int main()
{
Stack St1;
char Repeat='y';
int choice,temp , data;
clrscr();
while(Repeat == 'y' || Repeat == 'Y')
 {
    cout<<"\n 1. Push Operation";
    cout<<"\n 2. Pop Operation";
    cout<<"\n Enter your choice=";
    cin>> choice;
    switch(choice)
    {
    case 1: cout<< "\n Enter data for Push operation=";
          cin>> data;
          temp = St1.Push(data);
          if(temp==0)
          {
                    cout<<"\n Push Operation Unsuccessful";
            }
          else
          {
          cout<<"\n After Push operation::";
           St1.Display_Stack();
```

Space for learners:

```
        }
        break;
    case 2: temp = St1.Pop();
        if(temp==-99)
        {
                cout<<"\n Pop operation unsuccessful";
         }
        else
        {
        cout<<"\n"<<temp<<" is Poped from the Stack";
        cout<<"\n After Pop operation::";
        St1.Display_Stack();
         }
        break;
    default: cout<<"\n Wrong input";
    }
    cout<<"\n Input 'y' or 'Y' to repeat the operations=";
    cin>>Repeat;
     }
return 0;
}
```

Space for learners:

3.3.3 Implementation of Stack Using Linked List

In linked list implementation of stack, one special pointer is required to store the memory address of the first node in the linked list. Let us consider the name of this pointer as Top. Initially NULL value is assigned to Top which means that the stack is empty. So the underflow condition of linked list implementation of stack using C++ statement is presented below.

```
if ( Top = = NULL )
```

If the stack is empty then the Push operation on that stack can be performed by assigning the memory address of the new node to Top. Otherwise, Push operation is performed by inserting the node containing the new data at the first position in the linked list (Figure 3.2(b)).

If the underflow condition is false, then Pop operation on the stack can be performed by deleting the first node in the linked list. It means that the node pointed by Top is going to be deleted from the linked list. If there is no next node to the first node available in the linked list then it means that there is only one data available in the stack. In that case after the Pop operation, NULL value is assigned to Top. Otherwise, before deleting the current first node, Top is modified to point the second node so that after Pop operation, it becomes the new first node (Figure 3.2(c)).

Let us try to observe the following figures to understand the implementation of stack using Singly linked list.

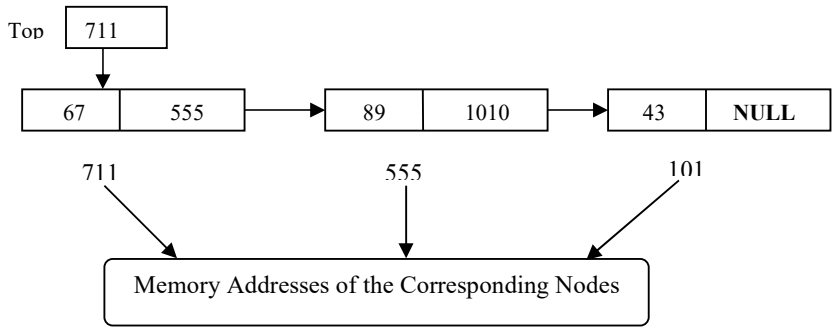


Figure 3.2(a): Stack Implemented Using Singly Linked List

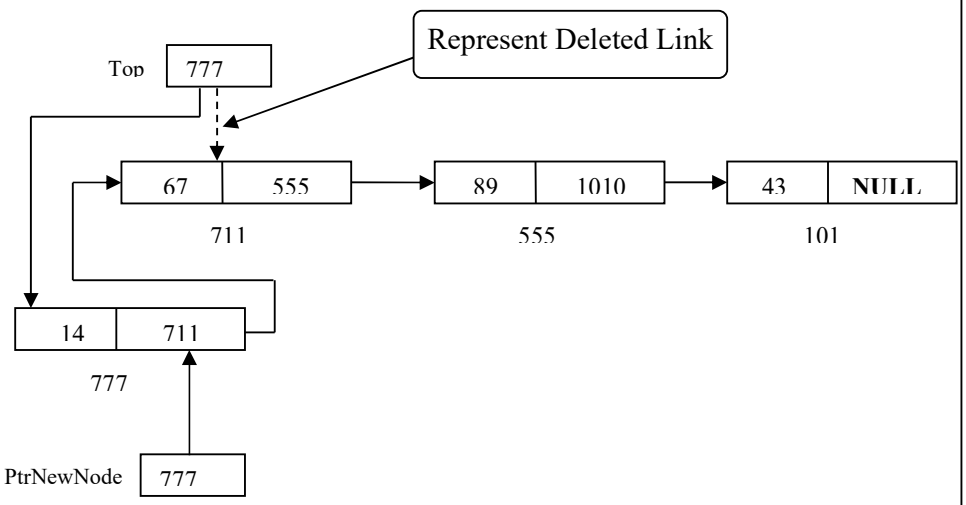


Figure 3.2(b): Stack (Represented In Figure3.2(a)) After Push Operation

Space for learners:

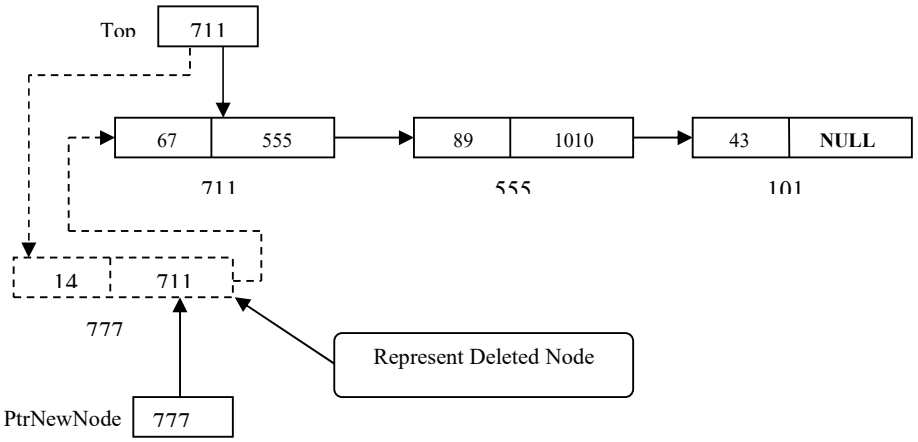


Figure 3.2(c): Stack (Represented In Figure3.2(b)) After Pop Operation

Space for learners:

Program 3.2:C++ program to implement stack using Singly linked list

```
# include <iostream.h>
# include <conio.h>

structStack_Node        // User defined data type to create node
{
        int data;
        structStack_Node *next;
};
typedefstructStack_Node Node;

class Stack             //Class to implement stack using Singly linked
list
{
private:
            Node *Top;

public:
        Stack()
            {
                Top = NULL;
            }
        int Push(int);
        int Pop();
```

```
        voidDisplay_Stack();
};

int Stack :: Push(int info)   // Push Operation
{
   Node *PtrNewNode;
PtrNewNode = new Node;
if(PtrNewNode == NULL)
    {
        cout<<"\n Memory Allocation Unsuccessful";
        return(0);
    }
else
    {

        PtrNewNode->data= info;
        PtrNewNode->next= NULL;
        if (Top == NULL)
             Top = PtrNewNode;
        else
         {
        PtrNewNode->next = Top;
            Top = PtrNewNode;
         }
        return(1);
    }
}

int Stack :: Pop()     //Pop Operation
{
int Deleted;
   Node *PtrDelete;
if( Top == NULL)     // Underflow condition
   {
        cout<<"\n Stack Underflow";
        return -99;
   }
else
   {
        Deleted = Top->data;
        PtrDelete=Top;
        Top = Top -> next;
        deletePtrDelete;
        return Deleted;
   }
}
```

Space for learners:

```
void Stack :: Display_Stack()   // Function to display stack data
{
Node *PtrTemp;
if(Top == NULL)
cout<<"\n Stack Underflow";
else
{
PtrTemp= Top;
cout<<"\n The Stack elements are::\n";
while(PtrTemp!=NULL)
{
        cout<<"\t"<<PtrTemp->data;
        PtrTemp=PtrTemp->next;
}
}
}

int main()
{
Stack St1;
char Repeat='y';
int choice,temp,info;
clrscr();
while(Repeat == 'y' || Repeat == 'Y')
 {
        cout<<"\n 1. Push Operation";
        cout<<"\n 2. Pop Operation";
        cout<<"\n Enter your choice=";
        cin>> choice;
        switch(choice)
        {
        case 1: cout<< "\n Enter data for Push operation=";
                cin>> info;
                temp = St1.Push(info);
                if(temp==0)
                 {
                        cout<<"\n Push Operation Unsuccessful";
                }
                else
                {
                cout<<"\n After Push operation::";
                 St1.Display_Stack();
                }
                break;
        case 2: temp = St1.Pop();
                if(temp==-99)
                {
```

Space for learners:

```
                cout<<"\n Pop operation unsuccessful";
            }
            else
            {
            cout<<"\n"<<temp<<" is Poped from the Stack";
            cout<<"\n After Pop operation::";
            St1.Display_Stack();
            }
            break;
        default: cout<<"\n Wrong input";
        }
        cout<<"\n Input 'y' or 'Y' to repeat the operations=";
        cin>>Repeat;
    }
return 0;
}
```

CHECK YOUR PROGRESS

1. Multiple choice questions:

A. Which of the following is not true in case of stack?

(i) Last element inserted will be removed first
(ii) First element inserted will be removed last
(iii) First element inserted will be removed first
(iv) Both (ii) and (iii)

B. Which of the following is an application of stack?

(i) Conversion of infix expression to its postfix expression
(ii) Implementation of CPU scheduling algorithms
(iii) Used in printer to print files
(iv) None of the above

C. In array implementation of stack using C++, if Top == 3 then it means____.

(i) the stack contains 2 data
(ii) the stack contains 3 data
(iii) the stack contains 4 data
(iv) None of the above

Space for learners:

D. Insertion operation in stack is termed as____.

(i)Push
(ii) Pop
(iii) Input
(iv) None of the above

E. Deletion operation in stack is termed as____.

(i) Push
(ii) Pop
(iii) Input
(iv) None of the above

F. Which of the data structure is used to implement Depth First Search traversal in Graph?

(i) Queue
(ii) Heap
(iii) Stack
(iv) None of the above

Space for learners:

3.4 QUEUE

Queue is defined as a linear data structure where the first data inserted will be removed first. It means that the order of insertion and deletion of a particular data must be same in case of queue. If a data is inserted to a queue as N^{th} data then it will be removed after removal of all (N-1) data that were inserted before the N^{th} data. Insertion of a new data is performed at one end in a queue which is called as Rear end. On the other hand deletion of existing data is performed at the other end in a queue that is termed as Front end. Queue is also termed as First-In First-Out (FIFO) data structure.

A
B
C

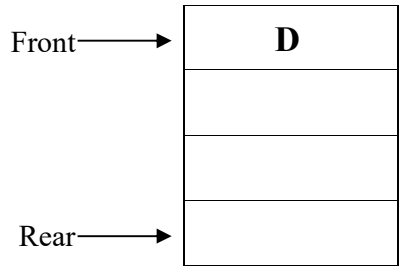


Figure 3.3(a): Diagrammatic Representation of a Queue with Four Existing Data

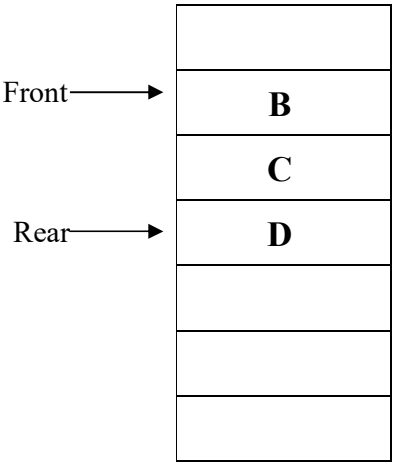


Figure 3.3(b): Queue After Deletion Operation

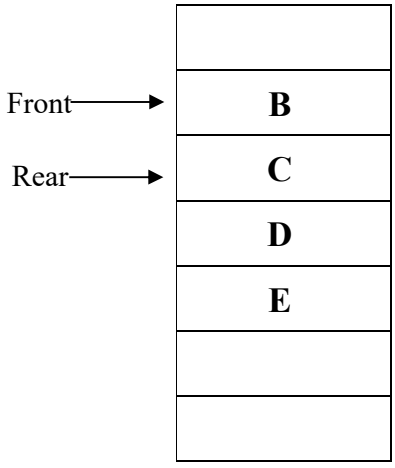


Figure 3.3(c): Queue After Insertion Operation

Space for learners:

In the above figure (Figure3.3(a)), a queue is represented which contains four data. Here, 'A' is the first inserted data in the queue and it is pointed by Front. Second, third and fourth inserted data are B, C and D respectively. So, Rear point to the last data that is 'D'. Now if delete operation is performed on the queue then the first data 'A' will be deleted and Front will point to the second data that is 'B' (Figure3.3(b)). So, now 'B' is the new first data and it will be deleted next. After the deletion operation, if insertion operation is performed on the queue then a new data is inserted to the queue that is 'E' and it is the new last data (Figure3.3(c)). At this point, Rear is modified and now it points to the new last data that is 'E'.

STOP TO CONSIDER

Queue can also be termed as Last In Last Out structure

Space for learners:

3.4.1 Applications of Queue

Queue is a very useful data structure as it is applied to implement a variety of important applications. A few of these applications are presented as follows.

1) Queue is used to implement process scheduling or CPU scheduling algorithms like First-Come First-Served, Round Robin etc.
2) Queue is used to implement printer spooler so that printer can print files in order to their arrival time.
3) Queue is used as buffer for devices like keyboards.
4) Queue is used to implement Breadth first traversal algorithm for graph traversal.
5) Queue is used to control congestions that are occurred in networks.
6) Queue is used in disk system to access files.
7) Queue is used to design different customer service related applications like ticket reservation systems.

3.4.2 Implementation of Queue Using Array

In this section, the implementation of queue using array is going to be discussed. In this implementation, at first, an array has to be declared to store data. Let us consider Queue_Arr as the array and

the size of this array is Queue_Size. It means the queue can be able to store at most Queue_Size numbers of data. Then two integer variables are also required for this implementation. Let us consider Front and Rearas the names of these variables. Initially both Front and Rearare assigned with -1 which means that the queue is empty. Front is used to store the subscript value of the cell in the array which store the data that will be deleted next. It means the current first data in the array is pointed by Front. On the other hand the subscript value of the cell storing the last data is stored in Rear.

To insert a new data into a queue, at first, overflow condition must be checked. If the overflow condition becomes true then it means that the queue doesn't have any empty space to insert a new data. The overflow condition using C++ statement is presented below.

```
if ( ( Front = = 0 )  && ( Rear = = Queue_Size-1) )
```

If overflow condition is not true then new data can be inserted in the empty cell that is next to the cell pointed by Rear. To access this empty cell, at first, Rear is incremented by one. Then using the array name and the incremented Rear, new data can be inserted to the appropriate cell in the queue. But if Rear stores the subscript value of the last cell in the array (Figure3.4(a)) then it mean that empty spaces are available in the left most portion of the array. In that case, at first data shifting must be performed to the left most portion of the array (Figure3.4(b)).After data shifting, empty spaces will be available to insert new data to the queue. Then new data is inserted to the cell next to the cell pointed by Rear.

	0	1	2	3	4	5	6	7	8
Queue_Arr				67	89	43	14	7	82

Front =3, Rear= 8

Figure 3.4(a): A Queue With No Empty Space Available Next To The Cell Pointed By Rear

Space for learners:

Space for learners:

	0	1	2	3	4	5	6	7	8
Queue_Arr	67	89	43	14	7	82			

Front =0, Rear= 5

Figure3.4(b): Queue After Data Shifting

To delete a data from the queue, at first underflow condition must be checked. If underflow condition is true then it means that the queue is empty. The underflow condition using C++ statement is presented below.

if (Front = = -1)

If underflow condition is not true then the data pointed by Front is considered to be deleted from the queue. If the value stored in Front is equal to the value stored in Rear then it means that there is only one data available in that queue. In that case, -1 is assigned to both Front and Rear. Otherwise, Front is incremented by one so that the data available in the next cell in the array becomes the new first data.

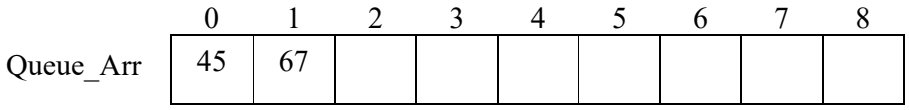
Let us try to observe the following diagrammatic presentations of a queue after insertion and deletion operations so that array implementation of queue can be visualized.

	0	1	2	3	4	5	6	7	8
Queue_Arr	45								

Front =0 , Rear=0

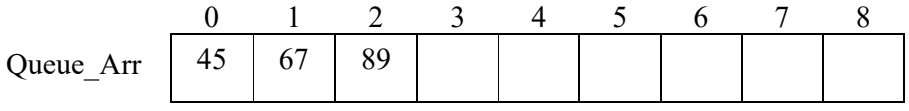
Figure3.5(a): QueueAfterFirst Insertion Operation

Space for learners:



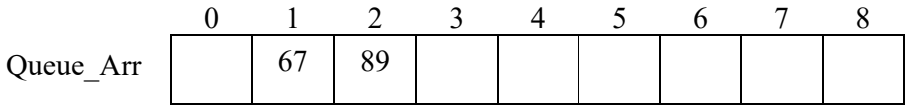
Front=0 , Rear=1

Figure3.5(b): Queue After Second Insertion Operation



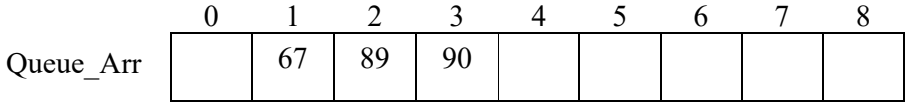
Front=0 , Rear=2

Figure3.5(c): Queue After Third Insertion Operation



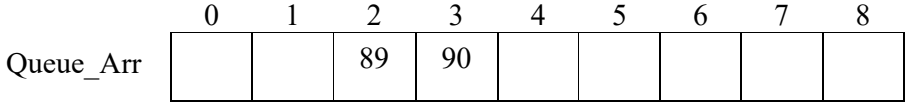
Front=1 , Rear=2

Figure3.5(d): Queue After First Deletion Operation



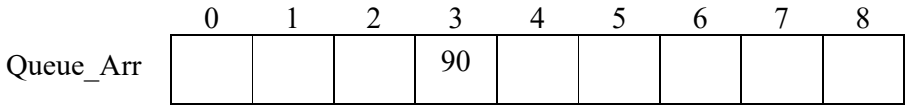
Front=1 , Rear=3

Figure3.5(e): Queue After Fourth Insertion Operation



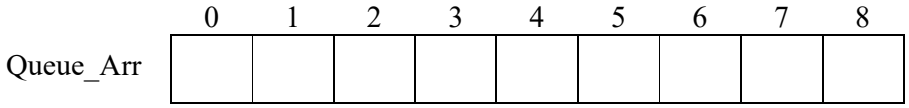
Front=2 , Rear=3

Figure3.5(f): Queue After Second Deletion Operation



Front=3 , Rear=3

Figure3.5(g): Queue After Third Deletion Operation



Front= -1 , Rear= -1

Figure3.5(h): Queue After Fourth Deletion Operation

Program 3.3:C++ program to implement queue using array

```
# include <iostream.h>
# include <conio.h>
# define Queue_Size 50

class Queue        // Class to implement queue using array
{
private:
        intQueue_Arr[Queue_Size];    // Array to store queue data
        int Front, Rear;
public:
        Queue()
            {
                Front = -1;
                Rear = -1;
            }
        int Insert(int);
        int Remove();
        voidDisplay_Queue();
};

int Queue :: Insert(int data)       //Insert Operation
{
int i;
```

Space for learners:

```
if( Front == 0 && Rear == Queue_Size -1 )    //  Overflow condition
    {
        cout<<"\n Queue Overflow";
        return(0);
    }
else
    {
        if( Front == -1)
         {
        Front = 0;
        Rear = 0;
         }
        else if( Rear == Queue_Size-1)
         {
        for( i =0; i <= Rear-Front; i++)
                Queue_Arr[i] = Queue_Arr[Front +i];          // Data
shifting
        Rear = Rear - Front + 1;
        Front =0;
         }
        else
        Rear =Rear+1;
        Queue_Arr[Rear]= data;
        return(1);
    }
}

int Queue :: Remove()        //  Delete Operation
{
int Deleted;
if( Front == -1)        // Underflow condition
{
        cout<<"\n Queue Underflow";
        return -99;
}
else
 {

        Deleted = Queue_Arr[Front];
```

Space for learners:

```
        if( Front == Rear)
        {
        Front = -1;
        Rear = -1;
        }
        else
        Front = Front + 1;
        return Deleted;
}
}

void Queue :: Display_Queue()     // Function to display queue data
{
int i;
if(Front == -1)
cout<<"\n Queue Underflow";
else
{
cout<<"\n The Queue elements are::\n";
for( i =Front;i <=Rear ; i++)
        cout<<"\t"<<Queue_Arr[i];
}
}

int main()
{
Queue Qu1;
char Repeat='y';
int choice,temp,data;
clrscr();
while(Repeat == 'y' || Repeat == 'Y')
 {
        cout<<"\n 1. Insert Operation";
        cout<<"\n 2. Delete Operation";
        cout<<"\n Enter your choice=";
        cin>> choice;
        switch(choice)
        {
        case 1: cout<< "\n Enter data for Insert operation=";
```

Space for learners:

```
                cin>> data;
                temp = Qu1.Insert(data);
                if(temp==0)
                   {
                        cout<<"\n Insertion Operation Unsuccessful";
                   }
                else
                   {
                cout<<"\n After Insertion operation::";
                     Qu1.Display_Queue();
                   }
                break;
        case 2: temp = Qu1.Remove();
                if(temp==-99)
                   {
                        cout<<"\n Delete operation unsuccessful";
                   }
                else
                   {
                cout<<"\n"<<temp<<" is deleted from the Queue";
                cout<<"\n After Delete operation::";
                Qu1.Display_Queue();
                   }
                break;
        default: cout<<"\n Wrong input";
        }
        cout<<"\n Input 'y' or 'Y' to repeat the operations=";
        cin>>Repeat;
    }
return 0;
}
```

3.4.2.1 Circular Queue

We have learnt that array implementation of queue requires data shifting for the insertion of new data when empty spaces are available in the queue but Rear is pointing to the last cell of the array (Figure3.6(a)). Data shifting is a very time consuming operation and that is why it should be avoided. To avoid it, the first cell of the array can be considered as the next cell to the last cell so

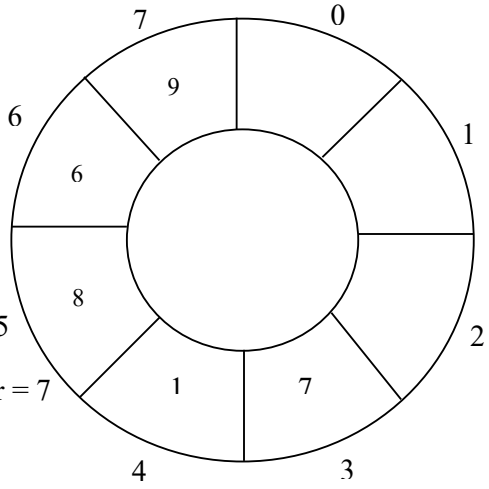
Space for learners:

that the array can be visualized as a circular structure (Figure3.6(b)). So, if Rear is pointing to the last cell in a queue then new data can be inserted in the first cell if it is empty and for this purpose 0 is assigned to Rear so that it will point to the first cell(Figure3.6(c) and Figure 3.6(d)). In this case, it can be observed that data shifting is not required to insert a new data to a queue if Rear is pointing to the last cell of the array. The queue that is implemented using array in this way is termed as circular queue. Let us try to observe the following figures to understand the basic concept of circular queue.

	0	1	2	3	4	5	6	7
Queue_Arr				77	14	82	67	91

Front = 3 , Rear = 7

Figure3.6(a): A Queue With Rear Is Pointing To The Last Cell Of The Array



Front = 3 , Rear = 7

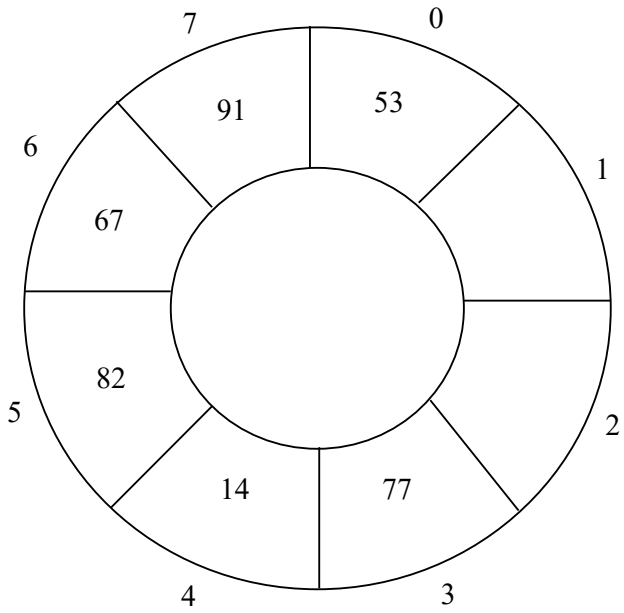
Figure3.6(b): Representation Of The Queue (Represented In Figure3.6(a)) As A Circular Structure

Space for learners:

	0	1	2	3	4	5	6	7
Queue_Arr	53			77	14	82	67	91

Front = 3 , Rear = 0

Figure3.6(c): After Insertion Operation In The Circular Queue (Represented In Figure 3.6(b))



Front = 3 , Rear = 0

Figure3.6(d): Representation of The Circular Queue (Represented In Figure 3.6(c)) As A Circular structure

In circular queue implementation, the values of Front and Rear are modified to their next values using Modulus (%) operator. We already know that Modulus (%) operator returns the reminder of adivision operation. The C++ statements to modify the values of Front and Rear are presented below.

```
Front = (Front + 1)% Queue_Size;
Rear = ( Rear  + 1 ) %Queue_Size;
```

Space for learners:

Here, Queue_Size is size of the array which is used to implement a circular queue. Let us consider the value of Queue_Size is 8 and Rear is pointing to the last cell. So the current value of Rear is 7.After modification of Rear using the above C++ statement, the value of Rear becomes 0. If we again modify Rearusing the above C++ statement then the value of Rear becomes 1. In this manner the value of Rear and Front can be modified to their next values.

We have already learnt that overflow condition must be checked before performing insertion operation on a queue.The overflow condition in case of circular queue using C++ statement is presented below.

if(Front == (Rear+1)% Queue_Size)

If overflow condition is not true then Rear is modified to its next value and new data is inserted in the empty cell that is pointed by Rear.

We have also learnt that underflow condition must be checked before performing delete operation on a queue.The underflow condition in case of circular queue using C++ statement is presented below.
if (Front = = -1)

If underflow condition is not true then the data pointed by Front is considered to be deleted from the queue. If the value stored in Front is equal to the value stored in Rear then it means that there is only one data available in that queue. In that case, -1 is assigned to both Front and Rear. Otherwise Front is modified to its next value.

Program 3.4:C++ program to implement circular queue

```
# include <iostream.h>
# include <conio.h>
# define CQueue_Size 50

classCQueue       // Class to implement circular queue
{
```

Space for learners:

```
private:
        intCQueue_Arr[CQueue_Size];   // Array to store queue data
        int Front, Rear;
public:
        CQueue()
            {
                Front = -1;
                Rear = -1;
            }
        int Insert(int);
        int Remove();
        voidDisplay_CQueue();
};

intCQueue :: Insert(int data)     // Insert Operation
{
int i;
if( Front == (Rear + 1)%CQueue_Size)     // Overflow condition
    {
        cout<<"\n Queue Overflow";
        return(0);
    }
else
    {
        if( Front == -1)
         {
            Front = 0;
            Rear = 0;
         }
        else
        Rear = ( Rear + 1 ) % CQueue_Size;    // Modification of
Rear value
        CQueue_Arr[Rear]= data;
        return(1);
    }
}

intCQueue :: Remove()      // Delete Operation
{
```

Space for learners:

```
int Deleted;
if( Front == -1)      // Underflow condition
   {
        cout<<"\n Queue Underflow";
        return -99;
   }
else
   {

        Deleted = CQueue_Arr[Front];
        if( Front == Rear)
        {
        Front = -1;
        Rear = -1;
        }
        else
        Front = (Front + 1) % CQueue_Size;    // Modification of
Front value
        return Deleted;
   }
}

voidCQueue :: Display_CQueue()   // Function to display queue data
{
int i;
if(Front == -1)
cout<<"\n Queue Underflow";
else
   {
cout<<"\n The Queue elements are::\n";
for(i =Front; i <=Rear ; i++)
        cout<<"\t"<<CQueue_Arr[i];
   }
}

int main()
{
CQueue CQu1;
char Repeat='y';
```

Space for learners:

```
int choice,temp,data;
clrscr();
while(Repeat == 'y' || Repeat == 'Y')
   {
        cout<<"\n 1. Insert Operation";
        cout<<"\n 2. Delete Operation";
        cout<<"\n Enter your choice=";
        cin>> choice;
        switch(choice)
        {
        case 1: cout<< "\n Enter data for Insert operation=";
                cin>> data;
                temp = CQu1.Insert(data);
                if(temp==0)
                   {
                        cout<<"\n Insertion Operation Unsuccessful";
                   }
                else
                   {
                cout<<"\n After Insertion operation::";
                CQu1.Display_CQueue();
                   }
                break;
        case 2: temp = CQu1.Remove();
                if(temp==-99)
                   {
                        cout<<"\n Delete operation unsuccessful";
                   }
                else
                   {
                cout<<"\n"<<temp<<" is deleted from the Queue";
                cout<<"\n After Delete operation::";
                CQu1.Display_CQueue();
                   }
                break;
        default: cout<<"\n Wrong input";
        }
        cout<<"\n Input 'y' or 'Y' to repeat the operations=";
        cin>>Repeat;
```

Space for learners:

```
    }
return 0;
}
```

3.4.3 Implementation of Queue Using Linked List

Queue can also be implemented using linked list. In case of linked list implementation, two special pointers are used to point the front and rear end of the queue. Let us consider Front and Rear as the names of these two pointers. Front is used to point the first data in the queue and Rear is used to point the last data in the queue. Initially NULL value is assigned to both Front and Rear which means that the queue is empty. So the underflow condition of linked list implementation of queue using C++ statement is presented below.

```
if ( Front = = NULL )
```

In linked list implementation of queue, if the queue is empty then both Front and Rear are modified to point the node containing the first data. Otherwise, the insertion operation is performed by inserting the node containing the new data at the last position. It means that the new node is inserted as a next node to the node pointed by Rear. After insertion of the new node, Rear is modified to point the new last node (Figure 3.7(b)).

If the underflow condition is false, then the deletion operation can be performed by deleting the first node in the queue. It means the node pointed by Front is going to be deleted from the linked list. If Front and Rear is pointing to the same node then it means that there is only one data available in the queue. In that case after deletion operation, NULL value is assigned to both Front and Rear. Otherwise, before deleting the current first node, Front is modified to point the second node so that after deletion operation, it becomes the new first node (Figure 3.7(c)).

Let us try to observe the following figures to understand the implementation of queue using Doubly linked list.

Space for learners:

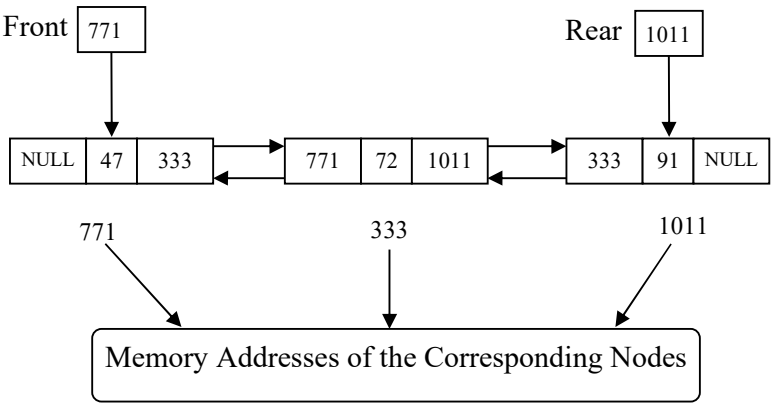


Figure3.7(a): Queue Implemented Using Doubly Linked List

Space for learners:

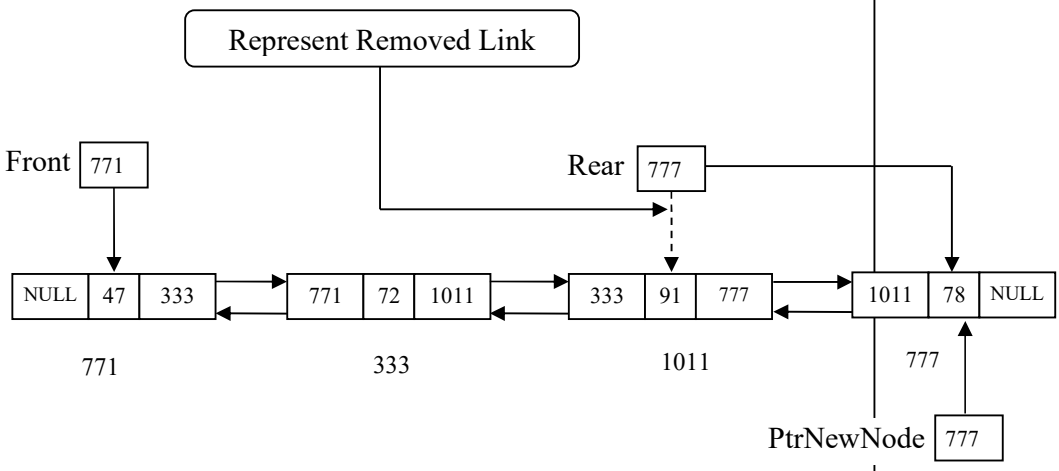


Figure 3.7(b):Queue (Represented In Figure3.7(a)) After Insertion Operation

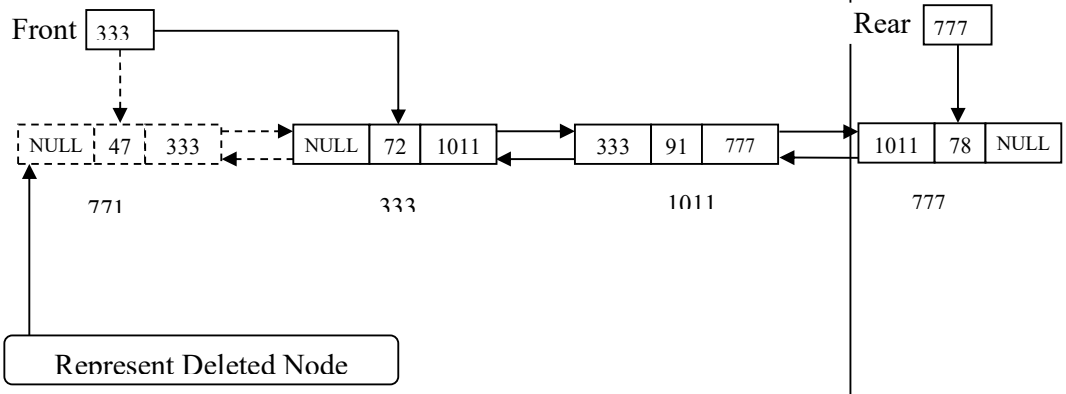


Figure 3.7(c): Queue (Represented In Figure3.7(b)) After Deletion Operation

Program 3.5:C++ program to implement queue using Doubly linked list

```
# include <iostream.h>
# include <conio.h>

structQueue_Node
{
        int data;
        structQueue_Node *prev;
        structQueue_Node *next;
};
typedefstructQueue_Node Node;

class Queue      // Class to implement queue using Doubly linked list
{
private:
            Node *Front,*Rear;

public:
        Queue()
            {
                Front = NULL;
                Rear = NULL;
            }
        int Insert(int);
        int Remove();
        voidDisplay_Queue();
};

int Queue :: Insert(int info)      // Insert Operation
{
   Node *PtrNewNode;
PtrNewNode = new Node;
if(PtrNewNode == NULL)
```

Space for learners:

```
    {
        cout<<"\n Memory Allocation Unsuccessful";
        return(0);
    }
else
    {

        PtrNewNode->data= info;
        PtrNewNode->prev=NULL;
        PtrNewNode->next= NULL;
        if (Front == NULL)
         {
            Front = PtrNewNode;
            Rear = PtrNewNode;
         }
        else
         {
        PtrNewNode->prev = Rear;
            Rear->next= PtrNewNode;
            Rear = PtrNewNode;
         }
        return(1);
    }
}

int Queue :: Remove()      // Delete Operation
{
int Deleted;
   Node *PtrDelete;
if( Front == NULL)
   {
        cout<<"\n Queue Underflow";
        return -99;
   }
else
   {
        PtrDelete=Front;
        if( Front ->next ==NULL)
          {
```

Space for learners:

```
            Front = NULL;
            Rear = NULL;
          }
        else
          {
            Front = Front->next;
            Front ->prev=NULL;
          }
        Deleted= PtrDelete->data;
        deletePtrDelete;
        return Deleted;
   }
}

void Queue :: Display_Queue()    // Function to display queue data
{
Node *PtrTemp;
if(Front == NULL)
cout<<"\n Queue Underflow";
else
{
PtrTemp= Front;
cout<<"\n The Queue elements are::\n";
while(PtrTemp!=NULL)
 {
        cout<<"\t"<<PtrTemp->data;
        PtrTemp=PtrTemp->next;
}
 }
}

int main()
{
 Queue Qu1;
char Repeat='y';
int choice,temp,info;
clrscr();
while(Repeat == 'y' || Repeat == 'Y')
{
```

Space for learners:

```
        cout<<"\n 1. Insert Operation";
        cout<<"\n 2. Delete Operation";
        cout<<"\n Enter your choice=";
        cin>> choice;
        switch(choice)
        {
        case 1: cout<< "\n Enter data for Insertion operation=";
                cin>> info;
                temp = Qu1.Insert(info);
                if(temp==0)
                   {
                        cout<<"\n Insertion Operation Unsuccessful";
                   }
                else
                   {
                cout<<"\n After Insertion operation::";
                     Qu1.Display_Queue();
                   }
                break;
        case 2: temp = Qu1.Remove();
                if(temp==-99)
                   {
                        cout<<"\n Delete operation unsuccessful";
                   }
                else
                   {
                cout<<"\n"<<temp<<" is deleted from the Queue";
                cout<<"\n After Delete operation::";
                 Qu1.Display_Queue();
                   }
                break;
        default: cout<<"\n Wrong input";
        }
        cout<<"\n Input 'y' or 'Y' to repeat the operations=";
        cin>>Repeat;
    }
return 0;
}
```

Space for learners:

3.5 PRIORITY QUEUE

Priority queue can be defined as a data structure where the deletion operation is depended upon the priority values associated with each data. Priority queue can be categorized into two types that are Max-priority queue and Min-priority queue. In case of Max-priority queue, the data with maximum priority value is going to be deleted first. On the other hand, in case of Min-priority queue, the data with minimum priority value is going to be deleted first. Priority queue is used to implement Priority Scheduling algorithm in operating system.

3.5.1 Implementation of Priority Queue

Priority queue can be implemented using array, linked list and heap.

A heap can be defined as a tree structure where each parent node holds higher key value than its child nodes or each parent node holds lower key value than its child nodes. If the key value of each parent node is greater than the key values of its child nodes then the heap is termed as Max-heap. On the other hand if the key value of each parent node is lower than the key values of its child nodes then the heap is termed as Min-heap. So the root node of a Max-heap holds the largest key value in the tree. On the other hand, the root node of a Min-heap holds the lowest key value in the tree. So Max-heap can be used to implement Max-priority queue and Min-heap can be used to implement Min-priority queue. In this implementation, at first a Max-heap or a Min-heap must be developed by considering the priority values associated with each data as the key values of each data or node. The deletion operation can be performed by replacing the data available at the root by a data available at any leaf node in the heap. Then the heap property must be established. The insertion operation can be performed by inserting new data to an empty space available in the heap and then establishing the heap property.

In case of linked list implementation of Priority queue, each node contains an extra field to store its priority value. Insertion of nodes are performed in the linked list in such a way that the nodes are available in that list in a sorted order based on the priority values associated with each node. So the deletion operation is performed

Space for learners:

by deleting the first node from the linked list. In case of Max-heap implementation, the nodes are available in the list in descending order based on their priority values. On the other hand, in case of Min-heap, the nodes are available in the list in ascending order based on their priority values.

STOP TO CONSIDER

Implementation of Priority queue using heap is the most efficient one.

Space for learners:

Program 3.6: C++ program to implement Max-priority queue using circular linked list

```
# include <iostream.h>
# include <conio.h>

structPQueue_Node
{
        int data;
        int priority;
        structPQueue_Node *next;
};
typedefstructPQueue_Node Node;


classPriority_Queue      // Class to implement Max-priority queue using circular linked list
{
private:
            Node *Front,*Rear;

public:
        Priority_Queue()
            {
                Front = NULL;
                Rear = NULL;
            }
        int Insert(int,int);
        int Remove();
        voidDisplay_Queue();
```

```
};

intPriority_Queue :: Insert(int info,intprio)
//Insert Operation
{
   Node *PtrNewNode,*PtrTemp,*PtrPrev;
PtrNewNode = new Node;
if(PtrNewNode == NULL)
   {
        cout<<"\n Memory Allocation
Unsuccessful";
        return(0);
   }
else
   {

        PtrNewNode->data= info;
        PtrNewNode->priority=prio;
        PtrNewNode->next= PtrNewNode;
        if (Front == NULL)
         {
            Front = PtrNewNode;
            Rear = PtrNewNode;
         }
        else
         {

        PtrTemp=Front;
        while(PtrNewNode->priority<=PtrTemp->priority
&&PtrTemp!=Rear)
             {
                PtrPrev=PtrTemp;
                PtrTemp=PtrTemp->next;
             }
        if(PtrTemp==Front)
             {
                PtrNewNode->next=Front;
                  Front = PtrNewNode;
                  Rear->next=Front;
             }
```

Space for learners:

```
        else if( PtrTemp==Rear &&PtrNewNode->priority<=PtrTemp->priority)
                {
                    PtrNewNode->next=Front;
                      Rear->next=PtrNewNode;
                      Rear=PtrNewNode;
                }
        else
                {
                    PtrNewNode->next=PtrTemp;
                    PtrPrev->next= PtrNewNode;
                }
         }
        return(1);
    }
}

intPriority_Queue :: Remove()        // Delete Operation
{
int Deleted;
   Node *PtrDelete;
if( Front == NULL)
   {
        cout<<"\n Queue Underflow";
        return -99;
   }
else
   {
        PtrDelete=Front;
        if( Front ->next ==Front)
          {
            Front = NULL;
            Rear = NULL;
          }
        else
          {
            Front = Front->next;
            Rear ->next=Front;
          }
        Deleted= PtrDelete->data;
```

Space for learners:

```
        deletePtrDelete;
        return Deleted;
}
}

voidPriority_Queue :: Display_Queue()  // Function to display
priority queue data
{
   Node *PtrTemp;
if(Front == NULL)
cout<<"\n Queue Underflow";
else
   {
PtrTemp= Front;
cout<<"\n The Queue elements with priority
values are::\n";
while(PtrTemp!=Rear)
 {
        cout<<"\tData:  "<<PtrTemp->data<<"    Pririty:"<<PtrTemp-
>priority;
        PtrTemp=PtrTemp->next;
     }
cout<<"\tData:  "<<PtrTemp->data<<"    Pririty:"<<PtrTemp-
>priority;
   }
}

int main()
{
Priority_Queue Qu1;
char Repeat='y';
int choice , temp,info,pvalue;
clrscr();
while(Repeat == 'y' || Repeat == 'Y')
   {
        cout<<"\n 1. Insert Operation";
        cout<<"\n 2. Delete Operation";
        cout<<"\n Enter your choice=";
        cin>> choice;
        switch(choice)
        {
```

Space for learners:

```
        case 1: cout<< "\n Enter data for Insertion
operation=";
                cin>> info;
                cout<<"\n Enter priority value of
the data=";
                cin>>pvalue;
                temp = Qu1.Insert(info,pvalue);
                if(temp==0)
                {
                        cout<<"\n          Insertion
Operation Unsuccessful";
                }
                else
                {
                cout<<"\n       After       Insertion
operation::";
                Qu1.Display_Queue();
                 }
                break;
        case 2: temp = Qu1.Remove();
                if(temp==-99)
                   {
                        cout<<"\n             Delete
operation unsuccessful";
                   }
                else
                   {
                cout<<"\n"<<temp<<" is deleted from the
Queue";
                cout<<"\n       After       Delete
operation::";
                Qu1.Display_Queue();
                   }
                break;
        default: cout<<"\n Wrong input";
        }
        cout<<"\n Input 'y' or 'Y' to repeat the
operations=";
        cin>>Repeat;
    }
return 0;
```

Space for learners:

}

Program 3.6:C++ program to implement Max-priority queue using Heap

```
#include<iostream.h>
#include<conio.h>
#define PQueue_Size 5
structPQdata
{
int data;
intpvalue;
};

classPQueue  // Class to implement Max-priority queue using heap
{
private:
        structPQdata PQ1[PQueue_Size];  // Array to store data and its priority value
        intHeap_Size;
public:
        PQueue()
            {
                Heap_Size=0;
            }
        voidDisplay_Data();
        voidCreate_Max_Heap();
        voidMax_Heap_Property(int);
        int Insert(int,int);
        int Remove();
};

voidPQueue:: Display_Data()  //Function to display priority queue data
{
int i;
if(Heap_Size==0)
cout<<"\n Queue Underflow";
else
```

Space for learners:

```
  {
cout<<"\n Queue elements with priority values are::\n";
for( i =1; i <=Heap_Size;i++)
{
        cout<<"\tData :"<<PQ1[i].data<<" Priority:
"<<PQ1[i].pvalue;
        cout<<"\n";
}
  }
}

voidPQueue:: Max_Heap_Property(inti )  // Function to establish
Max-heap property
{
int l,r,temp,largest;
 l=2*i;
 r= 2* i +1;
if(l<=Heap_Size&& PQ1[l].pvalue>PQ1[i].pvalue)
        largest=l;
else
        largest=i;
if(r<=Heap_Size&& PQ1[r].pvalue> PQ1[largest].pvalue)
largest =r;
if(largest!=i)
 {
        temp=PQ1[i].pvalue;
        PQ1[i].pvalue=PQ1[largest].pvalue;
        PQ1[largest].pvalue=temp;
        Max_Heap_Property(largest);
 }
}
voidPQueue:: Create_Max_Heap()  //Function to create a Max-heap
{
int i;
for(i =Heap_Size/2; i >=1;i--)
Max_Heap_Property(i);
}

intPQueue:: Insert(int info,intprio)  // Insert operation
```

Space for learners:

```
{

if(Heap_Size == PQueue_Size-1)    // Overflow condition
{
        cout<<"\n Priority Queue Overflow";
        return 0;
   }
else
   {

        Heap_Size = Heap_Size+1;
        PQ1[Heap_Size].data=info;
        PQ1[Heap_Size].pvalue=prio;
        Create_Max_Heap();
        return 1;
   }
}

intPQueue:: Remove()  // Delete Operation
{
int Deleted;
structPQdata temp;
if(Heap_Size==0)   // Underflow condition
 {
cout<<"\n Priority Queue Underflow";
return -99;
 }
else
 {
temp=PQ1[1];
PQ1[1]=PQ1[Heap_Size];
PQ1[Heap_Size]=temp;
Deleted=PQ1[Heap_Size].data;
Heap_Size=Heap_Size-1;
Max_Heap_Property(1);
return Deleted;
}
}
```

Space for learners:

```
int main()
{
PQueue PQu1;
char Repeat='y';
int choice,temp,data,prio;
clrscr();
while(Repeat == 'y' || Repeat == 'Y')
   {
        cout<<"\n 1. Insert Operation";
        cout<<"\n 2. Delete Operation";
        cout<<"\n Enter your choice=";
        cin>> choice;
        switch(choice)
        {
        case 1: cout<< "\n Enter data for Insert operation=";
                cin>> data;
cout<<"\n Enter priority value of the data=";
cin>>prio;
                temp = PQu1.Insert(data,prio);
                if(temp==0)
                {
                        cout<<"\n Insertion Operation Unsuccessful";
                }
                else
                 {
                cout<<"\n After Insertion operation::";
                  PQu1.Display_Data();
                 }
                break;
        case 2: temp = PQu1.Remove();
                if(temp==-99)
                 {
                        cout<<"\n Delete operation unsuccessful";
                }
                else
                 {
                cout<<"\n"<<temp<<" is deleted from the Queue";
                cout<<"\n After Delete operation::";
                PQu1.Display_Data();
```

Space for learners:

```
                }
                break;
        default: cout<<"\n Wrong input";
        }
        cout<<"\n Input 'y' or 'Y' to repeat the operations=";
        cin>>Repeat;
    }
return 0;
}
```

CHECK YOUR PROGRESS

2. Multiple choice questions:

A. Which of the following is not an application of queue?

(I) Implementation of Round Robin scheduling algorithm.
(ii) Used in printers to print files.
(iii) Used in network to control congestion
(iv) Used in reversal of strings.

B. Queue can also be termed as_____structure.

(i)First in First out
(ii)First in Last out
(iii)Last in First out
(iv)None of the above

C. In linked list implementation of queue, if Front == Rear then it means___.

(i) Queue underflow
(ii)Queue overflow
(iii)Only one data is available in the queue
(iv)None of the above

D. Insertion in queue is performed at _____.

(i)Front end
(ii)Rear end
(iii)Top end
(iv) Both A and B

Space for learners:

E. Which of the following can be used to implement queue?
(i) Array
(ii) Linked list
(iii) Heap
(iv) All of the above

F. Circular queue is implemented using_____.

(i)array
(ii)circular linked list
(iii)heap
(iv) None of the above

3. Fill in the blanks:

A. The overflow condition in circular queue is_____.
B. _____queue is used to implement Priority scheduling algorithm in operating system.
C. In linked list implementation of queue, the underflow condition is_____.
D. In queue, deletion is performed at _____end.
E. Implementation of Priority queue using ____ is the most efficient one.

Space for learners:

3.6 SUMMING UP

Stack is a linear data structure where the last data inserted will be removed first. In case of stack, the insertion and deletion operations are performed at the same end and it is termed as Top end. The insertion operation is termed as Push operation and deletion operation is termed as Pop operation.

Stack can be implemented using both array and linked list.

Queue is also a linear data structure where the first data inserted will be removed first. Queue is a double ended structure. The insertion operation is performed at one end that is called Rear end and on the

other hand the deletion operation is performed at the other end that is called as Front end.

Using both array and linked list, queue can be implemented.

In case of array implementation of queue, if the array is used as a circular structure by considering the first cell as the next cell to the last cell of the array then the queue is termed as Circular queue.

If the deletion of data is depended upon the priority value associated with each data in a queue then it is termed as Priority queue. There are two types of Priority queue available which are Max-priority queue and Min-priority queue. In Max-priority queue, the data with highest priority value will be deleted first. On the other hand, in Min-priority queue, the data with minimum priority value will be deleted first.

Priority queue can be implemented using array, linked list and heap. Implementation of Priority queue using heap is the most efficient one.

Space for learners:

3.7 ANSWERS TO CHECK YOUR PROGRESS

1 .A. (iii) , B. (i) , C. (iii) , D. (i) , E. (ii) , F.(iii)
2. A. (iv) , B. (i) , C. (iii) , D.(ii) , E. (iv) ,
   F. (i)
3. A. if (Front = = (Rear + 1) % Queue_Size
   B. Priority
   C. if (Front = = NULL)
   D. Front
   E. heap

3.8 POSSIBLE QUESTIONS

1) Define stack and queue. Write down some important applications of stack and queue.
2) Write a C++ program to implement stack using Doubly linked list.

3) Write a C++ program to implement queue using Circular linked list.
4) Explain the importance of circular queue with examples.
5) Write a C++ program to implement Min-priority queue using Singly linked list.

Space for learners:

3.9 REFERENCES AND SUGGESTED READINGS

1) Seymour Lipschutz : Data Structures With C, Tata McGraw-Hill
2) Ellis Horowitz, SartajSahni : Fundamentals of data structures, Computer Science Press
3) Yedidyah Langsam, Moshe J. Augenstein, Aaron M. Tenenbaum: Data structures using C and C++ , Prentice-Hall India

---×---

UNIT-4: BINARY TREE AND BINARY SEARCH TREE

Space for learners:

Unit Structure:

4.1 INTRODUCTION

Before going to enter in this chapter we need to study why tree data structure is required. Already we have studied various types of linear data structure like Stack, Queue, linked list etc. From memory point of view Linked list is a good data structure but its disadvantage is that it is a linear data structure. For searching elements in linked list, we need to visit all the elements of the linked list upto the searched element. This process is very slow and the complexity is O(n). If the element is not present in the list then we need to visit the entire element. In this chapter we will study the non-linear data structure in which data is organized in a hierarchical manner. A tree is such type of data structure where information retrieval and searching is very fast. Before going to the main topic we need to know the concept of tree.

Space for learners:

4.2 UNIT OBJECTIVES

In this Chapter we will study the concepts of the following:

- Understand the basic concept of Tree and basic terminology of tree.
- Know about Binary Tree, types of Binary, Representation of binary tree, traversal of binary tree.
- Know about Binary Search tree, traversal, searching Insertion and deletion in Binary Search Tree.

4.3 DEFINITION OF TREE

A tree is a finite set of nodes such that:

i) There is a starting node called root.

ii) The other nodes are partitioned into n>=0 disjoint sets $T_1,T_2…..T_n$, where each of this sets is also a tree. The sets $T_1,T_2…..T_n$ Are the subtrees of the root.

4.4 SOME BASIC TERMINOLOGY OF TREE

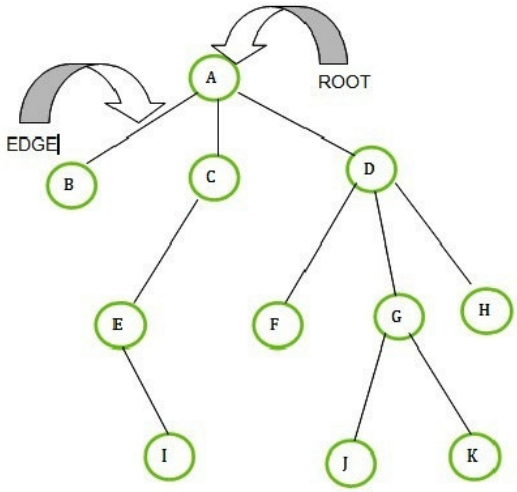
Space for learners:

Node:

Each element of tree is called a node. It may contain a value or condition.

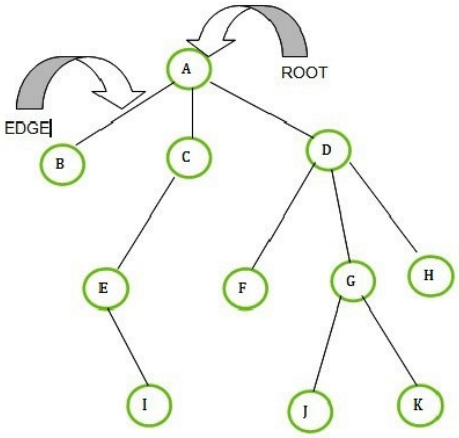
Root

It is specially designated node that does not have any parent node i.e The first node is called as Root Node. Every tree must have a root node. In a tree, there must be only one root node.



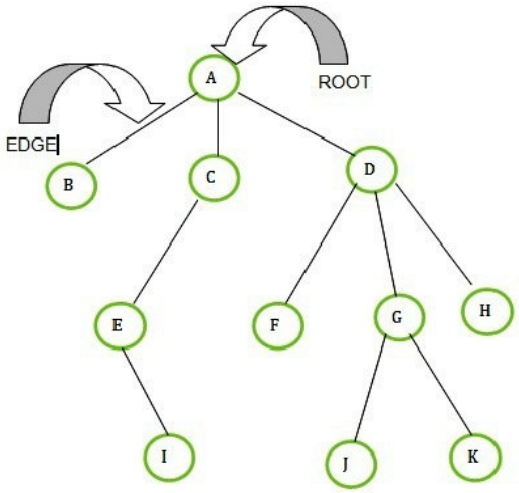
Edge

The connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



Parent Node

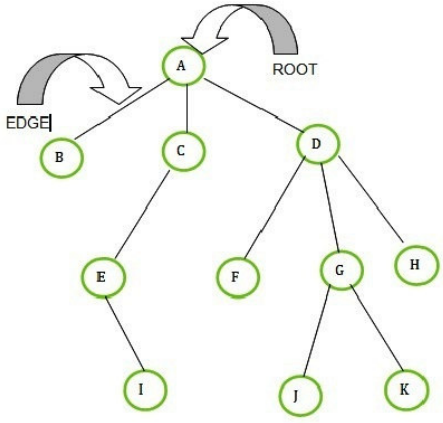
In a tree data structure, the immediate predecessor of a node is called as PARENT NODE. In simple words, the node which has a branch from it to any other node is called a parent node.



Here A,C,D,E,G are parent Nodes.

Child Node

In a tree data structure, all the immediate successor of nodes is called as CHILD Node. In other words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes.



Here B,C,D are children of A

E is the Children of C

Space for learners:

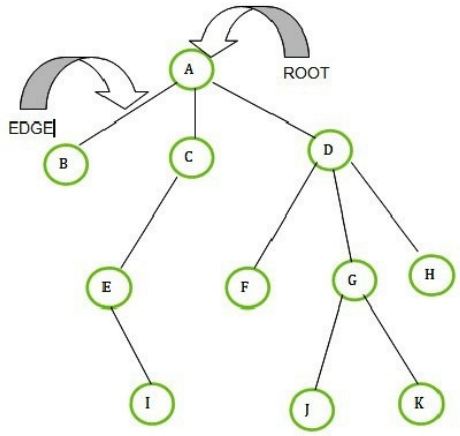
I is the children of E

F,G,H are children of D

J,K are children of G

Siblings

In a tree data structure, two or more nodes which have same parent are called Siblings.



Here B,C,D are Siblings

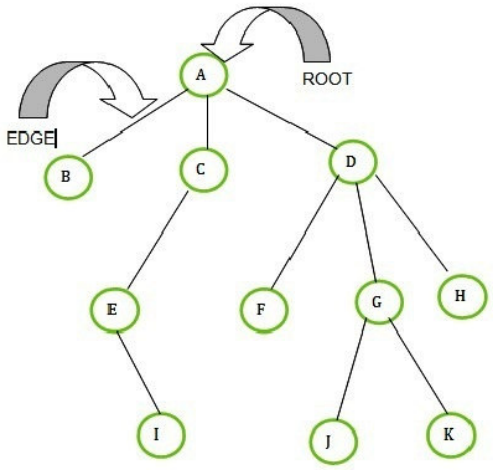
F,G,H are Siblings

J,K are Siblings

Leaf Node

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child.
In a tree data structure, the leaf nodes are also called as External Nodes or 'Terminal' node.

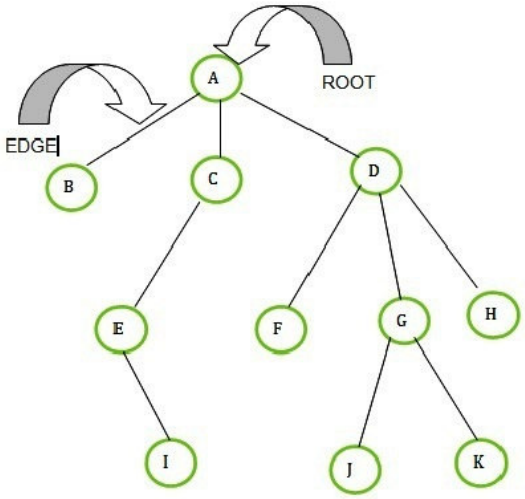
Space for learners:



Here B,I,F,J,K,H are Leaf nodes.

Internal Nodes

The node which has at least one child is called as INTERNAL Node. In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node. It is also called as 'Non-Terminal' nodes.



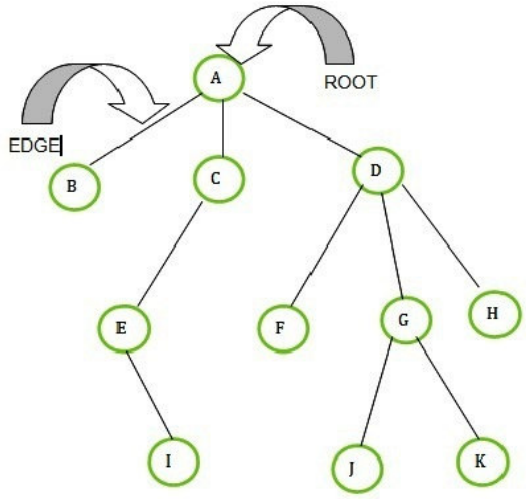
Here A,C,D,E,G are Internal Nodes.

Degree

In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In other words, the number of sub trees or children is also called its Degree.

Space for learners:

Space for learners:



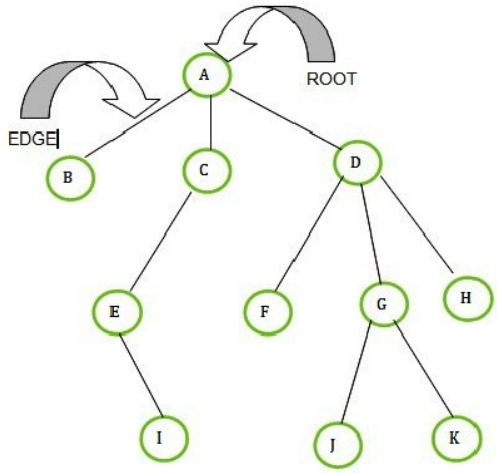
Here Degree of A is 3

Degree of C is 1

Degree of B is 0 etc.

Level

In a tree data structure, the distance of node from root is defined as Level of any node. Then, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on.



Here Level of A is 0

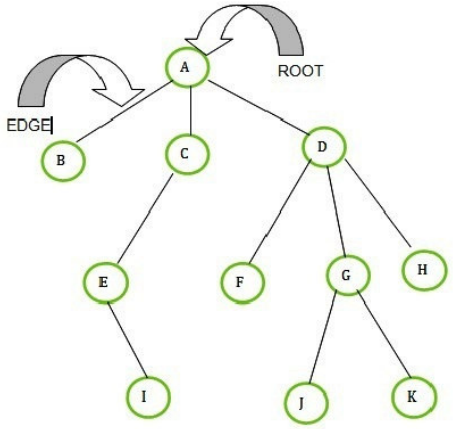
Level of B,C,D is 1

Level of E,F,G,H is 2

Level of I,J,K is 3

Height

In a tree data structure, the total number of level in a tree is the height of the tree. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



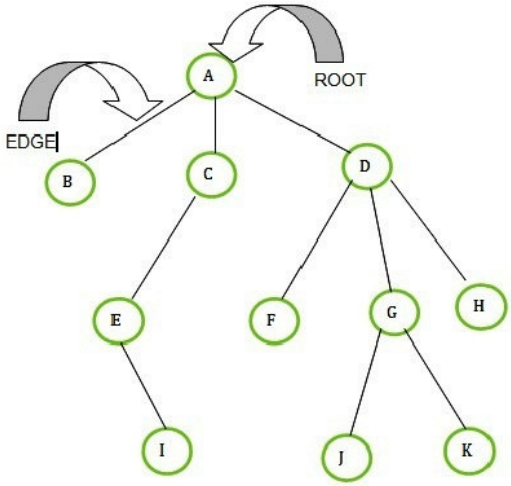
Here Height of K is 0

Height of H is 1 etc.

Height of the tree is 3.

Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between the two Nodes.

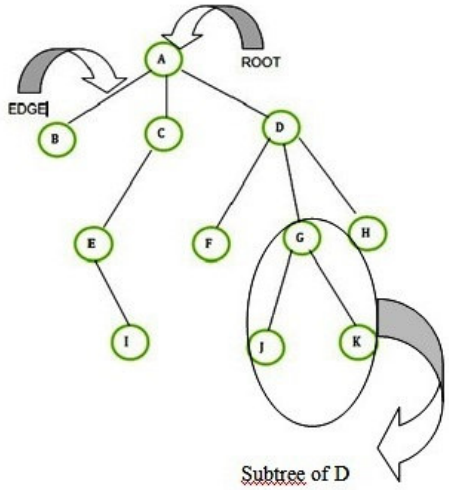


Here Path of A to K is A-D-G-K.

Space for learners:

Sub Tree

In a tree data structure, a tree may be divided into subtrees which can further be divided into subtrees. Each child from a node forms a subtree recursively.

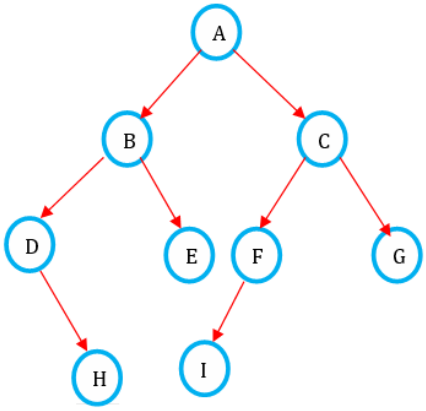


4.5 BINARY TREE

A tree can have any number of children. But a binary tree is a special type of tree in which no tree can have more than two children.

So, A Binary Tree can be defined as:

- Either empty tree or
- Consist of root node and remaining nodes are partitioned into two disjoint sets T_1 and T_2 and both are binary tree. T_1 and $T_2$are left and right subtree respectively.



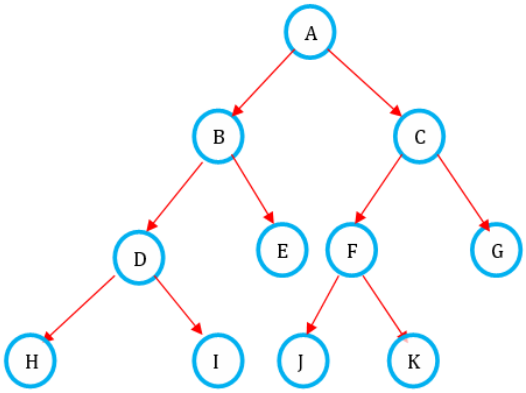
Space for learners:

4.5.1 Types of Binary Tree

There are different types of binary trees and they are:

4.5.1.1 Strictly Binary Tree

A Binary tree is strictly binary tree if each node in the tree is either a leaf node or should have exactly two children. That means every internal node must have exactly two children. Therefore, we can say that in strictly binary tree, there is no node with one child.



Property:

i) A Strictly binary tree with *n* non leaf nodes has *n+1* leaf node.

ii) A strictly binary tree with *n* leaf nodes always has *2n-1* nodes.

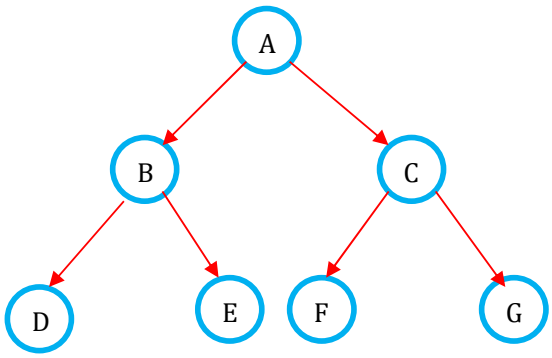
4.5.1.2 Full Binary tree

A Binary tree is defined as full binary tree in which all the nodes have 0 or two children. In other words, the full binary tree can be defined as a binary tree if all level has maximum number of nodes except the leaf nodes.

Here IF the number of any node is k, then the number of its left child is 2k, the number of its right child is 2k+1and the number of its parent is floor(k/2).

Space for learners:

Space for learners:



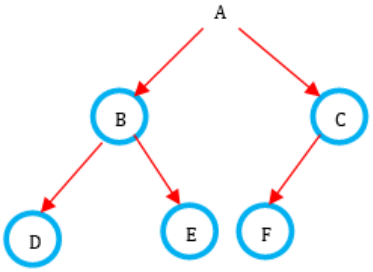
4.5.1.3 Complete Binary Tree

A complete binary tree is a binary tree when all the levels are completely filled i,e at every level all the nodes have exactly two children except the last level, which is filled from the left. Complete binary tree is also called as **Perfect Binary Tree.**

The complete binary tree is similar to the full binary tree except for the two differences which are given below:

i) The filling of the leaf node must start from the leftmost side.

ii) It is not mandatory that the last leaf node must have the right sibling.



The above tree is a complete binary tree, but not a full binary tree as node **F** does not have its right sibling.

4.5.2 Representation of Binary Tree

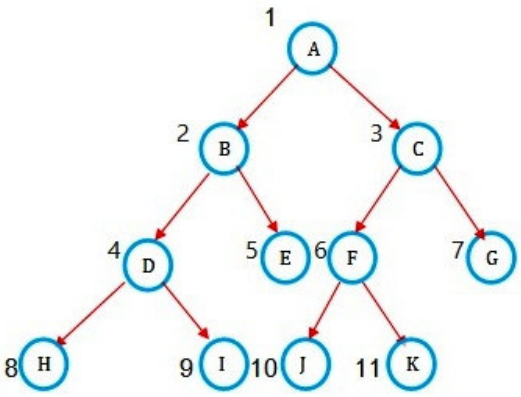
A binary tree data structure can be implemented by using two methods. Those methods are as follows:

- Array Representation
- Linked List Representation

Space for learners:

4.5.2.1 Array Representation of Binary Tree

It is also called sequential representation or linear representation or formula based representation. In this representation of a binary tree, we use one-dimensional array. Here it stores the tree data by scanning elements using level order fashion. So it stores nodes level by level. If some element is missing in a level, it left blank spaces for it.



The representation of the above tree is given below –

1	2	3	4	5	6	7	8	9	10	11
A	B	C	D	E	F	G	H	I	J	K

The index 1 is holding the root, it has two children B and C, they are placed at location 2 and 3. Few children may miss, so their place remains as blank.

In this representation we can easily get the position of two children of one node by using this formula −

$$child1 = 2 * parent$$

$$child2 = (2 * parent) + 1$$

To get parent index from child we have to follow this formula −

$$parent = [child / 2]$$

From execution point of view the sequential representation is efficient because we can calculate the index of parent and index of left and right children from the index of node. It is a static representation and the size of tree is restricted because of the limitation of array size. If array size is too small then overflow may occur and if array size is consider too large then space may be wasted. In this representation Insertion and deletion of nodes requires lot of movement of nodes in the array which consumes lot of time.

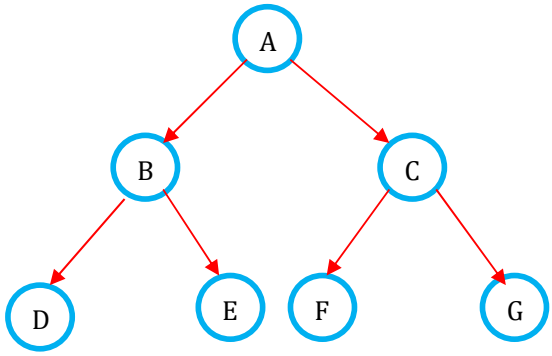
4.5.2.2 Linked List Representation of Binary Tree

Already we know that linked list representation is better than array representation from memory point of view because in this representation explicit pointers are used to link the nodes of the tree. Here, we will use a double linked list to represent a binary tree. In a double linked list, every node consists of a data part and two link parts. Left link part stores the address of left nodes and right link part store the address of right nodes. Data part stores the binary tree element. Addition and deletion of nodes requires less movement in comparison to array representation. Memory utilization is better in this representation.

The structure for tree node can be declared as:

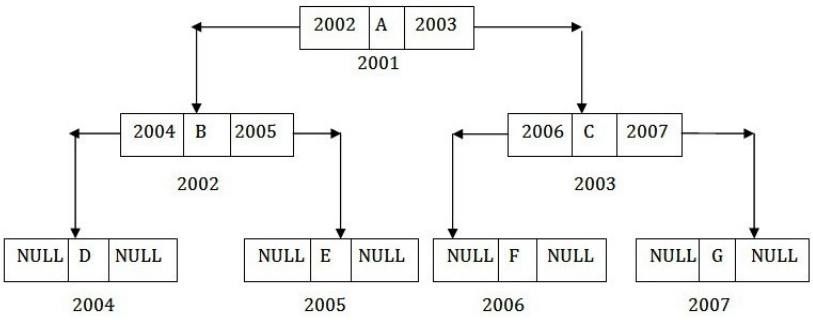
```
struct node
{   struct node *lchild;
    char data;
struct node * rchild;
};
```

From the below mentioned binary try we will represent it in doubly linked list:



Space for learners:

Space for learners:



In this representation uses dynamic memory allocation so we need not worry about the size of the tree. For addition and deletion, it uses less time as compared to array representation.

4.5.3 Traversal in Binary Tree

Traversal is a process to visit all the nodes of the tree exactly once. There are main three task in traversing- visiting the root node, traversing its left subtree and traversing its right subtree. For traversing, we always start from the root node. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

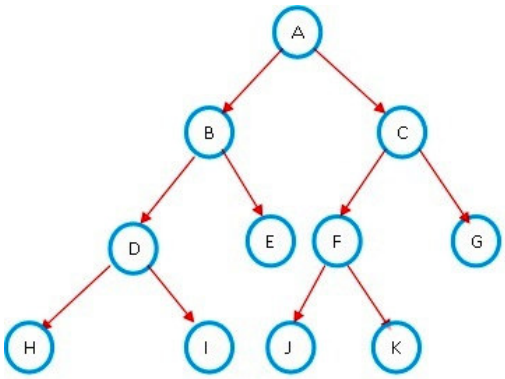
4.5.3.1 In-order Traversal (Root-Left-Right)

Here, the left subtree is visited first, then we visit the root and later we visit the right sub-tree. Every node may represent a subtree itself. In **in-order traversal**, the output will produce sorted key values in an ascending order.

Algorithm:

- Traverse the left sub-tree, (recursively call In-order (root -> left). (L)
- Visit the root (N)
- Traverse the right sub-tree, (recursively call In-order (root -> right). (R)

Space for learners:



In-order traversal of the above mentioned tree is:

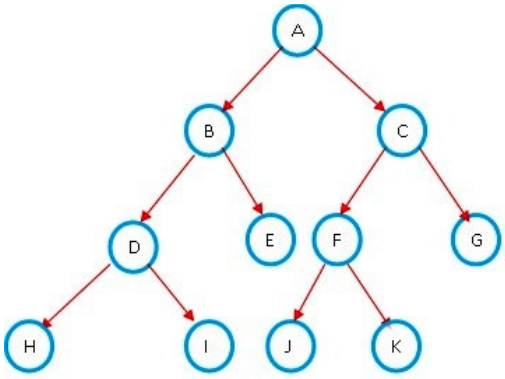
H-D-I-B-E-A-J-F-K-C-G

4.5.3.2 Pre-order Traversal (Root-Left-Right)

In this traversal method, the root node is visited first, then the left sub tree and finally the right subtree. Pre-order traversal can be used to make a prefix expression (Polish notation) from expression trees.

Algorithm:

- Visit the root (N)
- Traverse the left sub-tree, (recursively call In-order (root -> left). (L)
- Traverse the right sub-tree, (recursively call In-order(root -> right). (R)



Pre-order traversal of the above mentioned tree is:

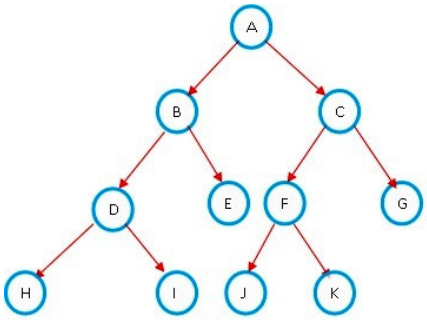
A-B-D-H-I-E-C-F-J-K-G

4.5.3.3 Post-order Traversal (Left-Right-Root)

In this traversal method, the left sub tree is visited first, then visit the right subtree, finally visit the root node. By Post-order traversal we can get the postfix expression of an expression.

Algorithm:

- Traverse the left sub-tree, (recursively call In-order(root -> left). (L)
- Traverse the right sub-tree, (recursively call In-order(root -> right). (R)
- Visit the root (N)



Post-order traversal of the above mentioned tree is:

H-I-D-E-B-J-K-F-G-C-A

4.6 BINARY SEARCH TREE

A binary search tree is an ordered tree which is specially use for the purpose of searching. Here, an element can be searched in average O(log N) time , where N is the number of nodes in the tree.

A Binary search tree is a binary tree in which-

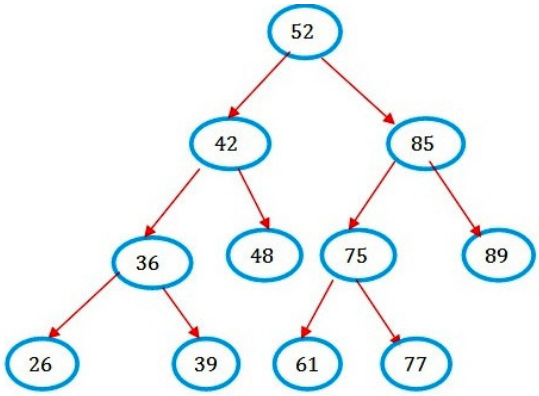
i. All the key values in the left subtree of root are less than the key value in the root.

Space for learners:

ii. All the key values in the right subtree of root are greater than the key value in the root.

iii. Left and right subtrees of root are also binary search tree.

The following is the example of Binary search tree:



4.6.1 Traversal in Binary Search Tree

Same methods are used here like Binary tree. From the above Binary search tree, we can find the following traversal methods:

In-order: 26-36-39-42-48-52-61-75-77-85-89

Pre-order: 52-42-36-26-39-48-85-75-61-77-89

Post-order: 26-39-36-48-42-61-77-75-89-85-52

Level order: 52-42-85-36-48-75-89-26-39-61-77

4.6.2 Searching in Binary Search Tree

We will start at the root node and compare the desired key with the key of root node. If the searched key is equal to the key in the root node, then the search is successful. If the searched key is less than the key of root node then we move to left subtree. If the searched key is greater than the key of root node then we move to right subtree. In this process, if we reach a NULL left child or NULL right child then the search is unsuccessful i.e desired key is not present in the tree. It is basically one kind of traversal in which at

Space for learners:

each step we will go either towards left or right and hence in at each step we discard one of the sub-trees.

Algorithm:

search (root, search_element)

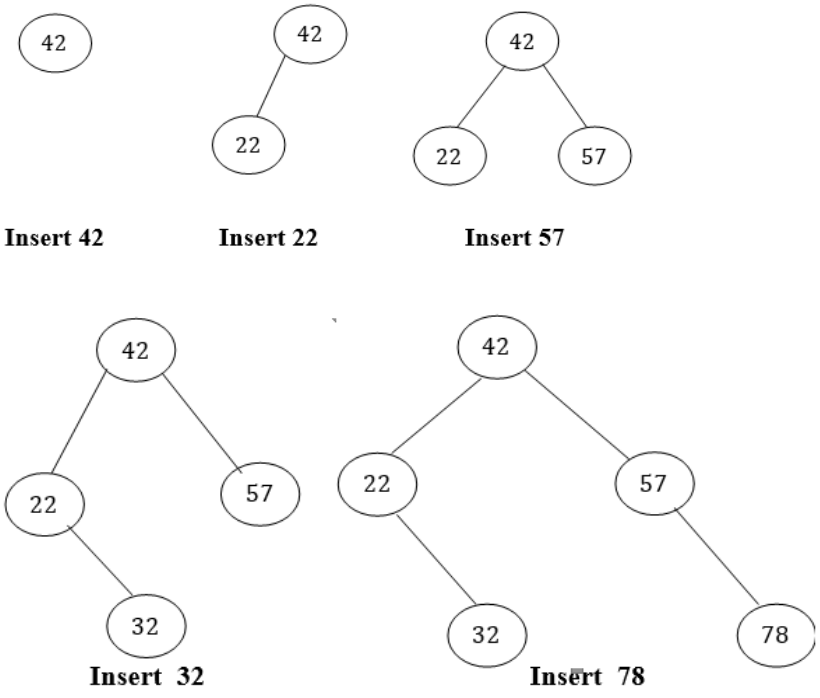
step 1: if root -> data = search_element or root = null
return root
else
if root <search_element
return search(root -> left, search_element)
else
return search(root -> right,search_element)
[end of if]
[end of if]

step 2: end

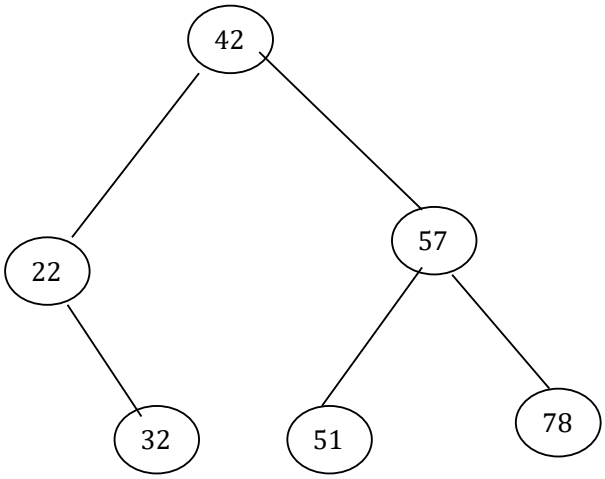
4.6.3 Creating a Binary Search Tree

Creating a Binary Search Tree from the key given below:

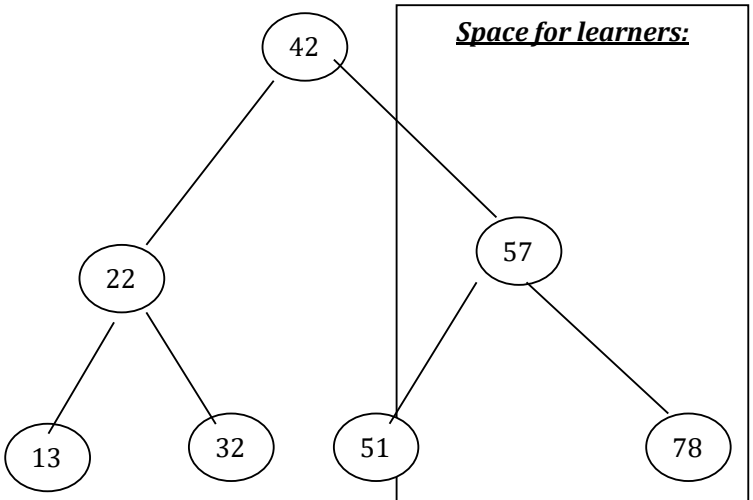
42, 22,57, 32,78,51,13, 54,82,69,29



Space for learners:

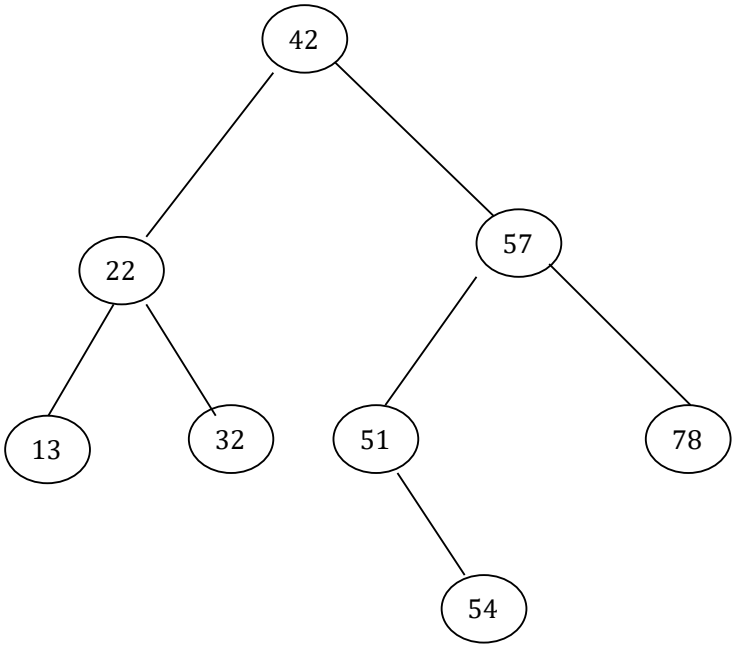


Insert 51

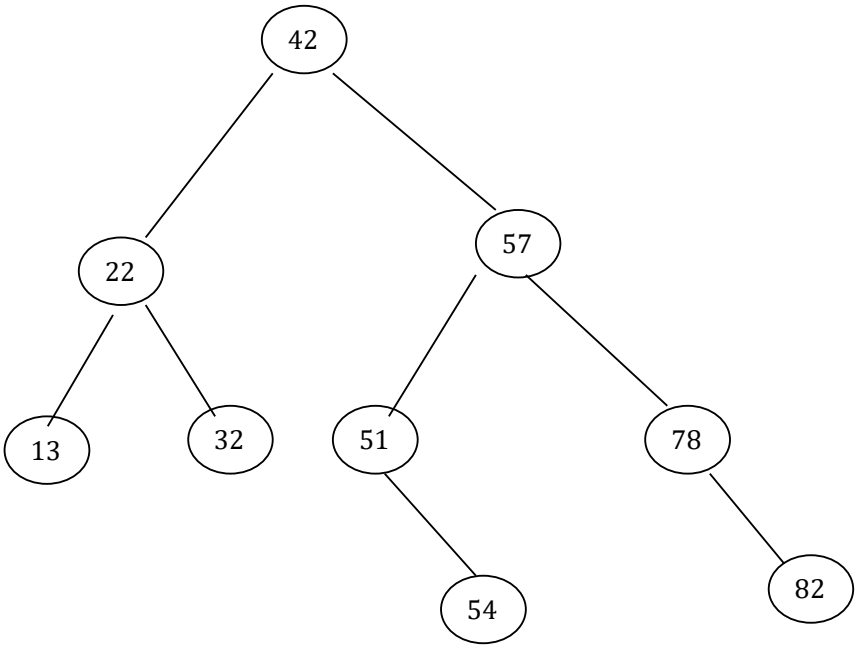


Insert 13

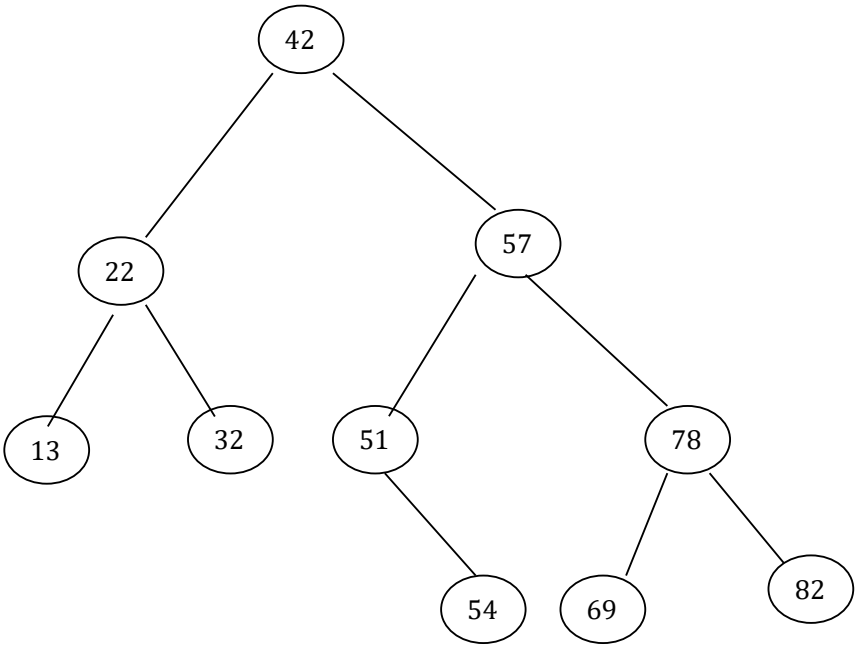
Space for learners:



Insert 54

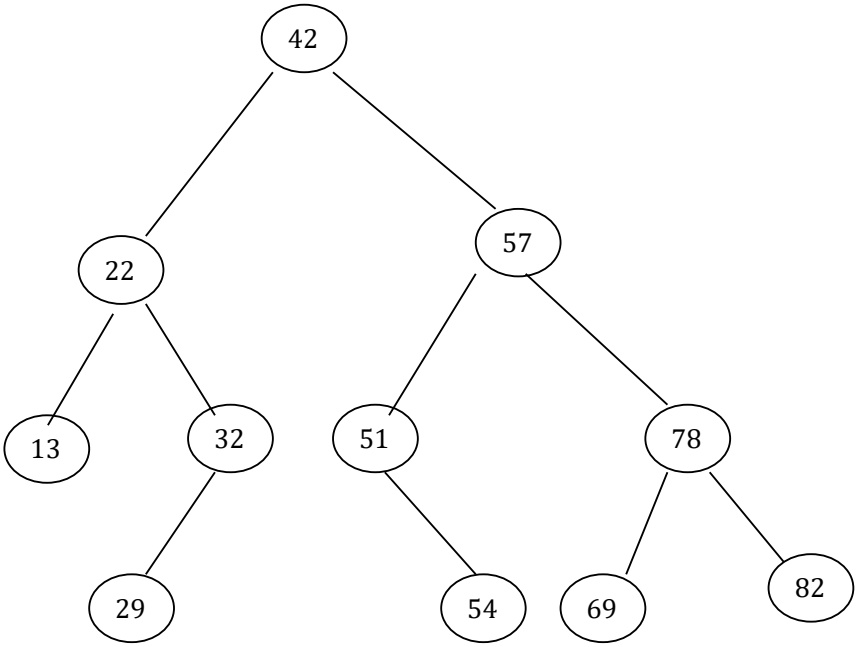


Insert 82



Insert 69

Space for learners:



Insert 29

Space for learners:

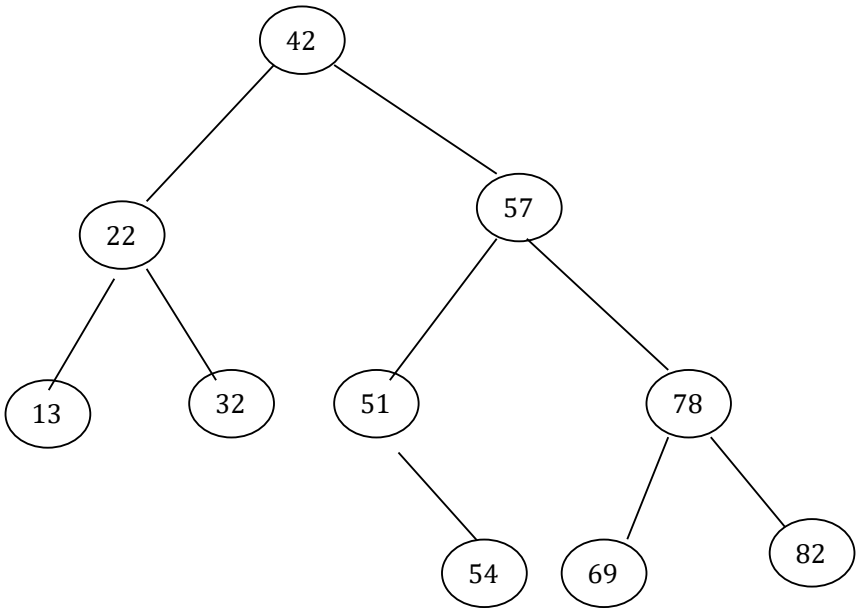
4.6.4 Insertion in Binary Search Tree

For inserting a node in Binary search tree means Insert a node in their appropriate position. Already we know that in binary search tree, element is lesser than the root node element will exist in the left side and element is greater than the root node will exist in the right side. When we insert an element in Binary search tree it should keep in mind that, it must not violate the property of binary search tree at each value. We will start from root and move down the tree. In each node we will compare with the insert element and take appropriate action. If the Insert item is equal to the node element, we will do nothing because duplicate element will not allow to insert in binary search tree. If the insert element is less than the node element then we will move to left node and perform this operation recursively. If the insert element is greater than the node element then we will move to right node and perform this operation recursively. We will insert the new key when we reach a NULL left or right node.

Insert (Tree, Insert_element)

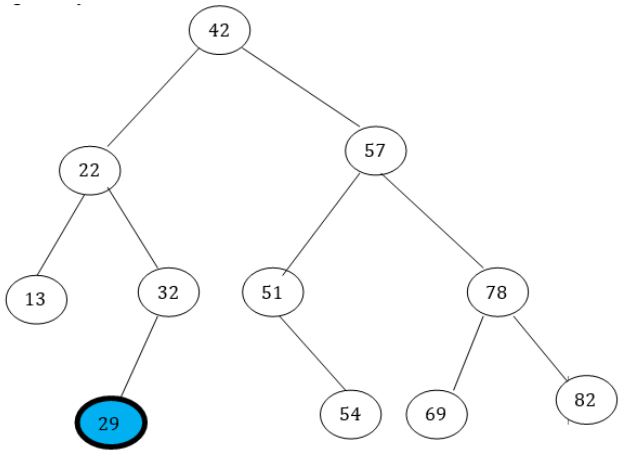
- **Step 1:** if tree = null
  allocate memory space for tree
  Set tree -> data = Insert_element
  Set tree -> left = tree -> right = null
  else
  if item < tree -> data
  Insert (tree -> left, Insert_element)
  else
  Insert (tree -> right, Insert_element)
  [end of if]
  [end of if]
- **Step 2:** end

Example: Suppose in the below mentioned Binary search tree, we would like to insert an element 29. Let's see the procedure.



First the insert element 29 will have to compare with the root element. Here 29 is less than the root element 42. So, we need to move to left child element. Now, here 29 is greater than 22. So, we move towards right child of the node. Now, again we compare with element 29 with right child element 32. Here we see that 29 is less than 32. So, we will move towards left child. But here no element exists. So, we will put this insert element in the left side. Now the resulting Binary search tree looks like:

Space for learners:



Space for learners:

4.6.5 Delete an Element from the Binary Search Tree

When we delete an element from Binary search tree it should keep in mind that, it must not violate the property of binary search tree at each value. There are three possible cases to delete an element in Binary search tree.

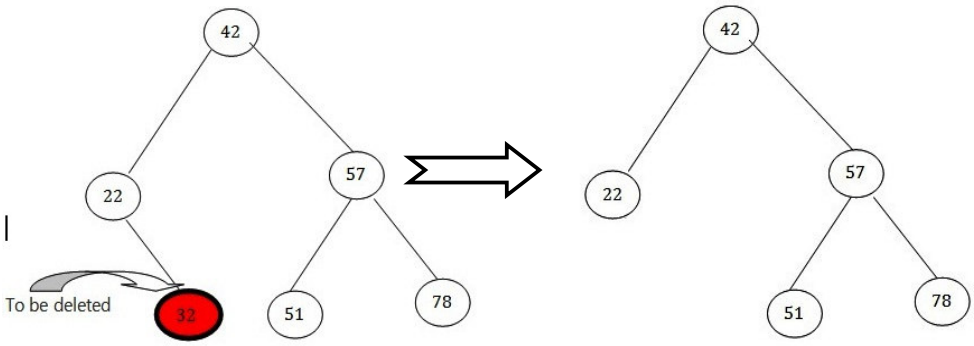
A. The node to be deleted has no child i.e., leaf node

B. The node to be deleted has exactly one child

C. The node to be deleted has exactly two children.

4.6.5.1 The Node to be Deleted has No Child

To delete a leaf node, replace the leaf node with the NULL and free the allocated space. When this procedure implemented by linked list, then left link of its parent is set to be NULL if the node is left child. If the node is right child of its parent then right link of its parent is set to NULL. After that free the memory space by free( ).



ace for learners:

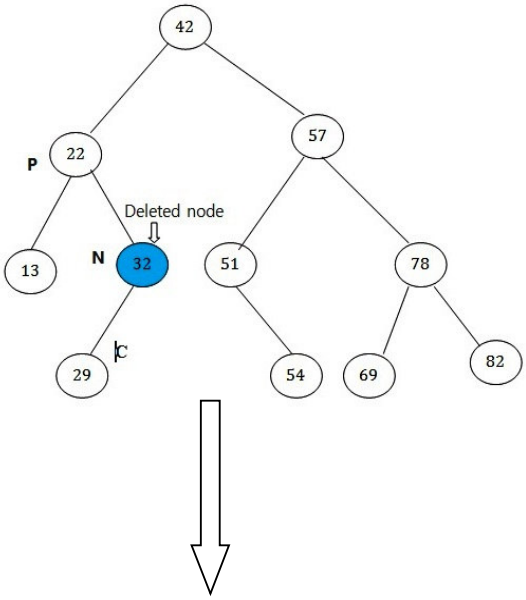
4.6.5.2 The Node to be Deleted has Exactly One Child

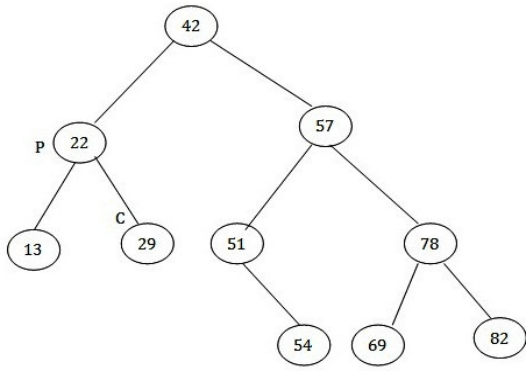
In this case, when we delete the node, the single child takes the position of the deleted node. After this, the memory space should be de-allocated using free( ).

Suppose N is the node to be deleted, P is the parent node and C is the child node.

If N is the left child of P, then the node C becomes left child of P after deletion.

If N is right child of P, then the node C becomes right child of P after deletion.





The Node 32 is to be deleted from the tree. Node 32 is right child of its parent 22. So, the single child 29 will take the position of 32.

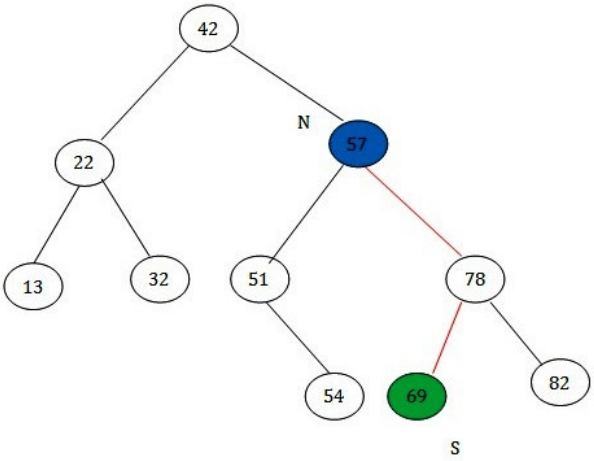
4.6.5.3 The Node to be Deleted has Exactly Two Children

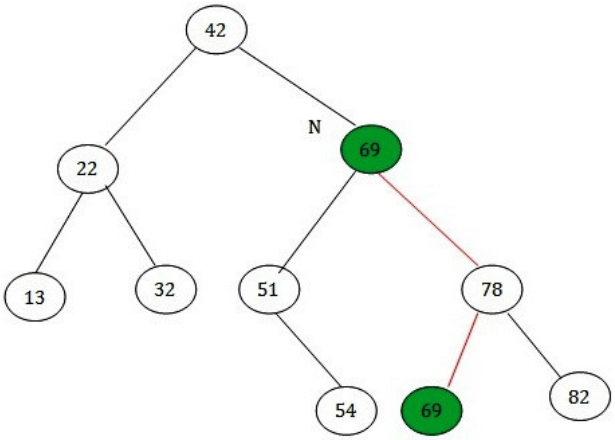
Here, first we need to find out the In-order successor of the deleted node. The data of the In-order successor copied to the deleted node and then delete the In-order successor from the tree. To find the In-order successor of a node N, first we directly move to the immediate right child of N and keep on moving left child till we find a node with no left child.

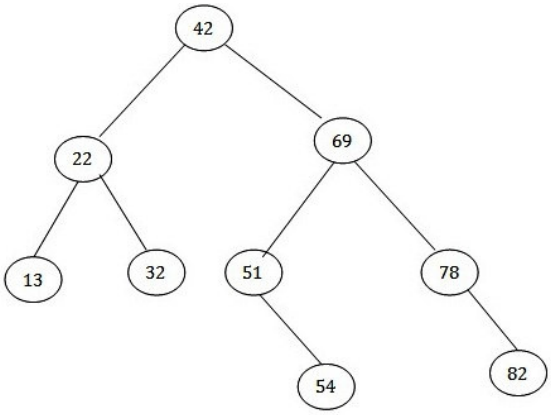
The point to be noted that, In-order successor is needed only when the right child is not empty. In this case, In-order successor can be obtained by finding the minimum value in the right child of the node.

Here node N having the key 57 is to be deleted. The in-order successor is node S having the key value 69. So, the data of node S is copied to node N and now node S is to be deleted from the tree. Now Node S can be deleted using case A because it has no child.

Space for learners:







Space for learners:

Space for learners:

CHECK YOUR PROGRESS

1. Multiple Choice Question

(i) The number of edges from the root to the node is called _____ of the tree.

a) Height
b) Length
c) Depth
d) Width

(ii) The number of edges from the node to the deepest leaf is called _____ of the tree.

a) Height
b) Depth
c) Length
d) Width

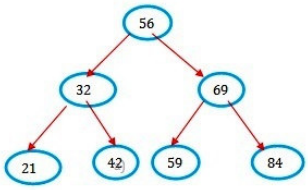
(iii) What is a full binary tree?

a) Each node has exactly zero or two children
b) Each node has exactly two children
c) All the leaves are at the same level
d) Each node has exactly one or two children

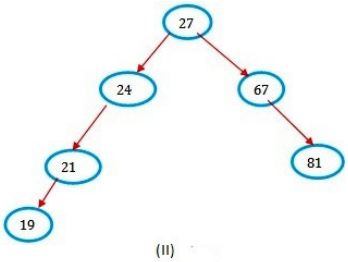
(iv) What is a complete binary tree?

a) Each node has exactly zero or two children
b) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from right to left
c) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right
d) A tree In which all nodes have degree

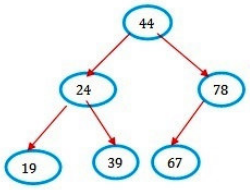
(v) Which of the following is not a binary search tree:



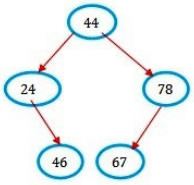
(I)



(II)



(III)



(IV)

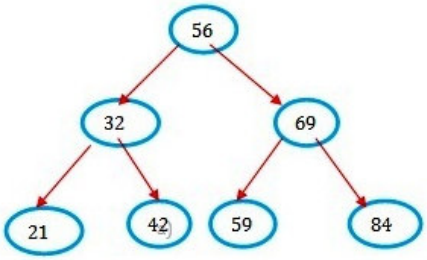
a) (I)

b) (II)

c) (III)

d) (IV)

(vi) If we delete 56 from the below Binary search tree after that what Parent --> Child pair does not occur in the tree?



a) 59-->69

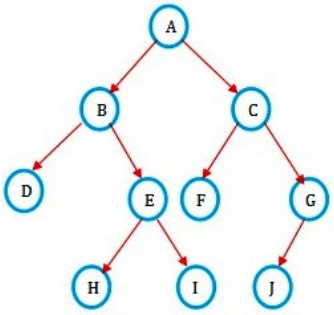
b) 32-->42

c) 59-->84

d) 32-->21

Space for learners:

(vii) The In-order traversal of the below mention Binary tree:



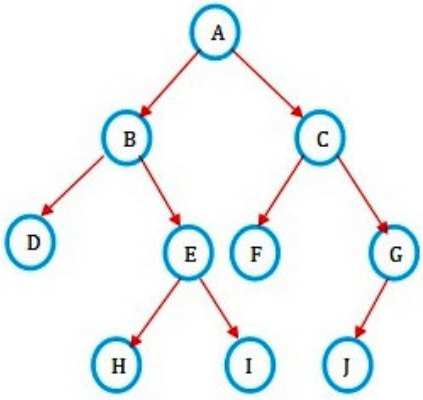
a) A-B-D-E-H-I-C-F-G-J

b) D-B-H-E-I-A-F-C-J-G

c) D-H-I-E-B-F-J-G-C-A

d) A-B-C-D-E-F-G-H-I-J

(viii) The Pre-order traversal of the below mention Binary tree:



a) A-B-D-E-H-I-C-F-G-J

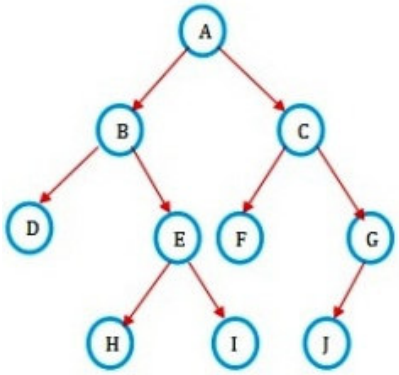
b) D-B-H-E-I-A-F-C-J-G

c) D-H-I-E-B-F-J-G-C-A

d) A-B-C-D-E-F-G-H-I-J

(ix) The Post-order traversal of the below mention Binary tree:

Space for learners:



a) A-B-D-E-H-I-C-F-G-J

b) D-B-H-E-I-A-F-C-J-G

c) D-H-I-E-B-F-J-G-C-A

d) A-B-C-D-E-F-G-H-I-J

(x) What is the specialty about the In-order traversal of a binary search tree?

a) It traverses in a non-increasing order

b) It traverses in an increasing order

c) It traverses in a random fashion

d) It traverses based on priority of the node

2. State True or False:

(i) In a tree data structure, all the immediate successor of nodes is called as Parent Node.

(ii) In a tree data structure, the total number of children of a node is called as Degree of that Node.

(iii) A Binary tree can have any number of children.

(iv) A Binary tree is strictly binary tree if each node in the tree is either a leaf node or should have exactly two children.

(v) The full binary tree can be defined as a binary tree if all level has maximum number of nodes except the leaf nodes.

Space for learners:

3. Fill up the Blanks and Answer

(i) For searching elements in linked list, the complexity is_____.

(ii) In a tree, the immediate predecessor of a node is called as _____.

(iii) Every tree must have a _____node.

(iv) _____ is specially designated node that does not have any parent node.

(v) The node of a tree which does not have a child is called as _____ Node.

(vi) In _____traversal the left subtree is visited first, then we visit the root and later we visit the right sub-tree. Every node may represent a subtree itself.

(vii) In _____traversal, the root node is visited first, then the left sub tree and finally the right subtree.

(viii) In _____Traversal (Left-Right-Root), the left sub tree is visited first, then visit the right subtree, finally visit the root node. By Post-order traversal we can get the postfix expression of an expression.

Space for learners:

4.7 SUMMING UP

- A tree is a hierarchical data structure defined as a collection of nodes.
- Each element of tree is called a node. It may contain a value or condition.
- Root is specially designated node that does not have any parent node.
- The connecting link between any two nodes is called as EDGE.
- In a tree data structure, the immediate predecessor of a node is called as PARENT NODE.

- All the immediate successor of nodes is called as CHILD Node.
- Two or more nodes which have same parent are called Siblings.
- In a tree which does not have a child is called as LEAF Node.
  - The node which has at least one child is called as INTERNAL Node.
- The total number of children of a node is called as DEGREE of that Node. In other words, the number of subtrees or children is also called its Degree.
    - The distance of node from root is defined as Level of any node.
    - The total number of level in a tree is the height of the tree.
    - The sequence of Nodes and Edges from one node to another node is called as PATH between the two Nodes.
    - A tree may be divided into subtrees which can further be divided into subtrees. Each child from a node forms a subtree recursively.
- A binary tree is a special type of tree in which no tree can have more than two children.
- A Binary tree is strictly binary tree if each node in the tree is either a leaf node or should have exactly two children.
- A Binary tree is defined as full binary tree in which all the nodes have 0 or two children.
- A complete binary tree is a binary tree when all the levels are completely filled i.e., at every level all the nodes have exactly two children except the last level, which is filled from the left.
- A binary tree data structure can be implemented by using Array and Linked List
- Traversal is a process to visit all the nodes of the tree exactly once. There are main three task in traversing- visiting the

Space for learners:

root node, traversing its left subtree and traversing its right subtree. For traversing always we start from the root node.

- In In-order Traversal the left subtree is visited first, then we visit the root and later we visit the right sub-tree. Every node may represent a subtree itself.
- In Pre- order traversal, the root node is visited first, then the left sub tree and finally the right subtree.
- In Post-order Traversal (Left-Right-Root), the left sub tree is visited first, then visit the right subtree, finally visit the root node. By Post-order traversal we can get the postfix expression of an expression.
- A Binary search tree is a binary tree in which all the key values in the left subtree of root are less than the key value of the root and all the key values in the right subtree of root are greater than the key value of the root. Left and right subtrees of root are also binary search tree.
- In Searching in Binary search Tree need start at the root node and compare the desired key with the key of root node. If the searched key is equal to the key in the root node, then the search is successful. If the searched key is less than the key of root node then we move to left subtree. If the searched key is greater than the key of root node then we move to right subtree. In this process, if we reach a NULL left child or NULL right child then the search is unsuccessful.
- In Insertion in Binary Search Tree, we need start from root and move down the tree. In each node we will compare with the insert element and take appropriate action. If the Insert item is equal to the node element, we will do nothing because duplicate element will not allow to insert in binary search tree. If the insert element is less than the node element then we will move to left node and perform this operation recursively. If the insert element is greater than the node element then we will move to right node and perform this operation recursively. We will insert the new key when we reach a NULL left or right node.
- To delete a leaf node, replace the leaf node with the NULL and free the allocated space. When this procedure

Space for learners:

implemented by linked list, then left link of its parent is set to be NULL if the node is left child. If the node is right child of its parent, then right link of its parent is set to NULL. After that free the memory space by free().

- When we delete a node from Binary search tree who have exactly one child, the single child takes the position of the deleted node. After this, the memory space should be de-allocated using free() .
- When we delete a node from the Binary search tree that have exactly two children, first we need to find out the In-order successor of the deleted node. The data of the In-order successor copied to the deleted node and then delete the In-order successor from the tree.

Space for learners:

4.8 ANSWERS TO CHECK YOUR PROGRESS

1. (i) c) , (ii) a) , (iii) a), (iv) c) , (v) d) , (vi) c) , (vii) b) ,(viii) a) , (ix) c) , (x) b)

2. (i) False , (ii) True , (iii) False , (iv) True , (v) True.
3. (i) O(n)
   (ii) Parent Node
   (iii) Root
   (iv) Root
   (v) Leaf
   (vi) In-order
   (vii) Pre-order
   (viii) Post-order

4.9 POSSIBLE QUESTIONS

1. Define tree in a data Structure?
2. Why tree data structure is useful?
3. What is leaf node of a tree?

4. Define Height of a tree.
5. Why root node important for a tree?
6. What is Strictly Binary tree?
7. What is Full Binary tree?
8. What is complete binary tree?
9. Define In-order traversal of a tree?
10. Define Pre-order traversal of a tree?
11. Define Post order traversal of a tree?
12. What is Binary search Tree?
13. What is the main characteristic of a binary search tree?
14. Explain the two types of representation of Binary tree with diagram.
15. Explain various kinds of traversal techniques of binary tree with example.
16. Write the algorithm to search an element in a binary search tree?
17. Write down the insertion procedure of Binary search tree.
18. Write down the working procedure to delete an element from the binary search tree where-
    a. The node to be deleted has no child i.e leaf node.
    b. The node to be deleted has exactly one child.
    c. The node to be deleted has exactly two children.
19. Write a function to check whether a binary tree is binary search tree or not.
20. Write a program to display all the leaf nodes of binary tree.

Space for learners:

4.10 REFERENCES AND SUGGESTED READINGS

- Srivastava, Suresh Kumar, and Deepali Srivastava. *Data Structures through C in depth*. BPB publications, 2004.
- Thareja, Reema. *Data structures using C*. Oxford University Press, Inc., 2011.

BLOCK II:
DICTIONARY ADT AND SORTING AND SELECTION TECHNIQUES

UNIT 1: INTRODUCTION TO SEARCH TREES

Space for learners:

Unit Structure:

1.1 INTRODUCTION

In this unit, you will learn the fundamental aspects pertaining to various search trees used in data structure. Search trees can be used to support dynamic sets, i.e. data structures that change during lifetime, where an ordering relation among the keys is defined. They support many operations, such as SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE. The time that an operation takes depends on the height h of the tree.

1.2 UNIT OBJECTIVES

Space for learners:

After going through this unit, you will be able to:

- *understand* the fundamental concepts of search tree.
- *know* different types of search trees.
- *understand* the functionality of different search trees.
- *how to* perform various operations on search trees.

1.3 DEFINITION OF SEARCH TREES

A search tree (Figure 1.1) is a tree data structure used for locating specific keys from a set of elements. In order for a tree to function as a search tree, the key for each node must be greater than any keys in sub-trees on the left, and less than any keys in sub-trees on the right The main advantage of search trees is their efficient search time given the tree is reasonably balanced, which is to say the leaves at either end are of comparable depths. Various search-tree data structures exist, several of which also allow efficient insertion and deletion of elements, which operations then have to maintain tree balance. Search Trees allow efficient searching of ordered data by implementing Ordered Dictionary ADT. It provides flexible mechanism for storing and retrieving data.

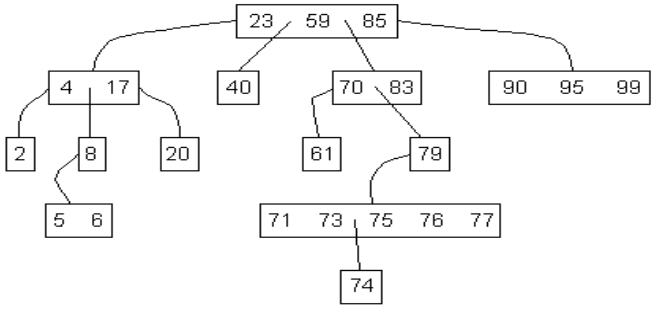


Figure 1.1: An example of a search tree

1.4 TYPES OF SEARCH TREES

There are many different search trees are available and they are different in nature. The most commonly used search trees are

- Binary Search Tree (BST)
- AVL Tree
- B Tree
- (a,b)- Tree
- Red-Black Tree
- Splay Tree
- Ternary Search Tree

Space for learners:

1.4.1 BINARY SEARCH TREE (BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the following properties –

- The value of the key of the left sub-tree is less than the value of its root node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its root node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

Left subtree (keys) < node (key) ≤ Right subtree (keys)

A diagrammatic representation of a Binary Search Tree (Figure 1.2) is given below-

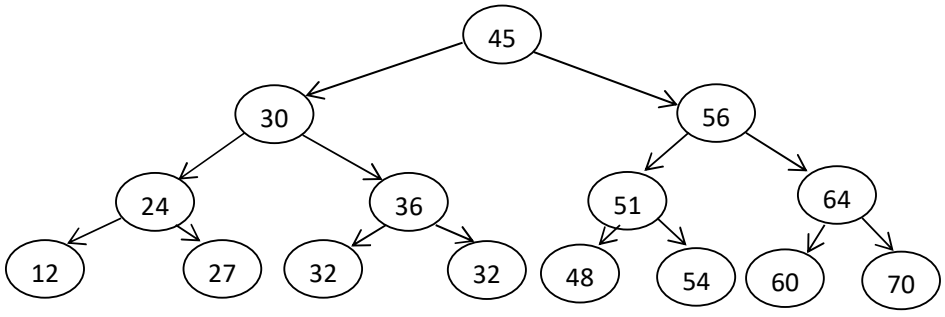


Figure 1.2: Example of a Binary Search Tree

Space for learners:

The following operations are normally performed in a BST-

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

The time complexity of BST is as follows-

Operation	Average Case	Worst Case
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Search	O(log n)	O(n)

1.4.2 AVL TREE

AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by two Soviet scientists Georgy Adelson-Velskii, and Evgenii Landis and hence given the short form as AVL tree or Balanced Binary Tree. The AVL was published in 1962 in a paper *"An algorithm for the organization of information"* in the proceedings of USSR Academy of Sciences (in Russian), which was later translated to English by Myron J. Ricci in *Soviet Mathematics – Doklady.*

An AVL tree can be defined as follows-

Let T be a non-empty binary tree with T_L and T_R as its left and right sub-trees. The tree is height balanced if:

- T_L and T_R are height balanced

- $h_L - h_R <= 1$, where $h_L - h_R$ are the heights of T_L and T_R

The Balance factor of a node in a binary tree can have value 1, -1, 0, depending on whether the height of its left sub-tree is greater, less than or equal to the height of the right sub-tree.

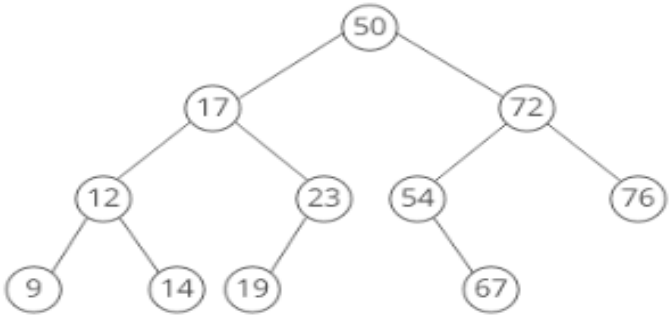


Figure 1.3: Balanced Binary Search Tree

In the above figure the value of BF is 1 hence this is a balance binary tree.

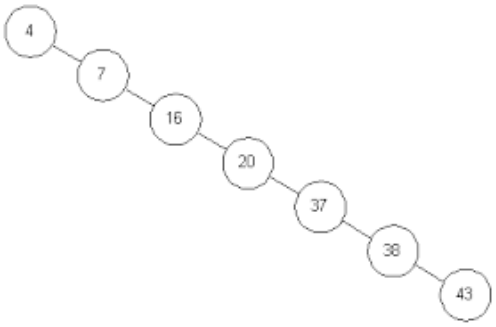


Figure 1.4: Unbalanced Binary Search Tree

In the above figure the value of BF is less than -1 hence this is an unbalance binary tree.

The detail operation on AVL is discussed in the next unit.

1.4.3 B TREE

B Tree is a self-balancing data structure based on a specific set of rules for searching, inserting, and deleting the data in a faster and memory efficient way. In order to achieve this, the following rules are followed to create a B Tree.

Space for learners:

A B-Tree is a special kind of tree in a data structure. In 1972, this method was first introduced by McCreight, and Bayer named it Height Balanced m-way Search Tree. It helps you to preserves data sorted and allowed various operations like Insertion, searching, and deletion in less time. B-trees are designed to work well on magnetic disks or other direct-access secondary storage devices. B-trees are similar to red-black trees, but they are better at minimizing disk I/O operations.

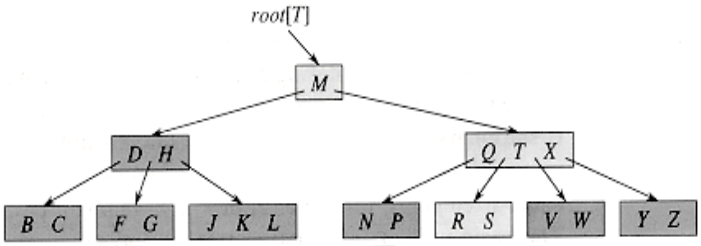


Figure 1.5: A B-tree whose keys are the consonants of English.

An internal node x containing n[x] keys has n[x] + 1 children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

B-trees generalize binary search trees in a natural manner. Figure 1.5 shows a simple B-tree. If a B-tree node x contains $n[x]$ keys, then x has $n[x] + 1$ children. The keys in node x are used as dividing points separating the range of keys handled by x into $n[x] + 1$ sub-ranges, each handled by one child of x. When searching for a key in a B-tree, we make an $(n[x] + 1)$-way decision based on comparisons with the $n[x]$ keys stored at node x.

There are many different technologies available for providing memory capacity in a computer system. The ***primary memory*** (or ***main memory***) of a computer system typically consists of silicon memory chips, each of which can hold 1 million bits of data. This technology is more expensive per bit stored than magnetic storage technology, such as tapes or disks. A typical computer system has ***secondary storage*** based on magnetic disks; the amount of such secondary storage often exceeds the amount of primary memory by several orders of magnitude.

Space for learners:

Space for learners:

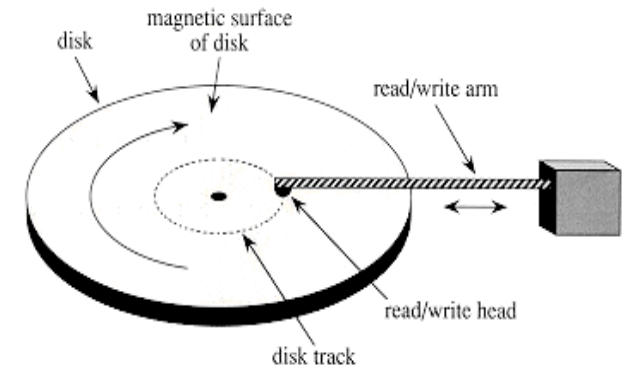


Figure 1.6: A Typical Disk Drive

Figure 1.6 shows a typical disk drive. The disk surface is covered with a magnetizable material. The read/write head can read or write data magnetically on the rotating disk surface. The read/write arm can position the head at different distances from the center of the disk. When the head is stationary, the surface that passes underneath it is called a ***track.*** The information stored on each track is often divided into a fixed number of equal-sized ***pages***; for a typical disk, a page might be 2048 bytes in length. The basic unit of information storage and retrieval is usually a page of information--that is, disk reads and writes are typically of entire pages. The ***access time***--the time required to position the read/write head and to wait for a given page of information to pass underneath the head--may be large (e.g., 20 milliseconds), while the time to read or write a page, once accessed, is small. The price paid for the low cost of magnetic storage techniques is thus the relatively long time it takes to access the data. Since moving electrons is much easier than moving large (or even small) objects, storage devices that are entirely electronic, such as silicon memory chips, have a much smaller access time than storage devices that have moving parts, such as magnetic disk drives. However, once everything is positioned correctly, reading or writing a magnetic disk is entirely electronic (aside from the rotation of the disk), and large amounts of data can be read or written quickly.

Often, it takes more time to access a page of information and read it from a disk than it takes for the computer to examine all the information read. For this reason, in this chapter we shall look separately at the two principal components of the running time:

Space for learners:

- The number of disk accesses
- The CPU (computing) time.

The number of disk accesses is measured in terms of the number of pages of information that need to be read from or written to the disk. We note that disk access time is not constant--it depends on the distance between the current track and the desired track and also on the initial rotational state of the disk. We shall nonetheless use the number of pages read or written as a crude first-order approximation of the total time spent accessing the disk.

In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk pages that have changed. Since the B-tree algorithms only need a constant number of pages in main memory at any time, the size of main memory does not limit the size of B-trees that can be handled.

We model disk operations in our pseudo code as follows-

Let x be a pointer to an object. If the object is currently in the computer's main memory, then we can refer to the fields of the object as usual: $key[x]$, for example. If the object referred to by x resides on disk, however, then we must perform the operation DISK-READ(x) to read object x into main memory before its fields can be referred to. (We assume that if x is already in main memory, then DISK-READ(x) requires no disk accesses; it is a "noop.") Similarly, the operation DISK-WRITE(x) is used to save any changes that have been made to the fields of object x. That is, the typical pattern for working with an object is as follows-

```
1    . . .
2    x ← a pointer to some object
3    DISK-READ(x)
4    operations that access and/or modify the fields of x
5    DISK-WRITE(x)    ▷ omitted if no fields of x were changed.
6    the operations that access but do not modify fields of x
```

7 ...

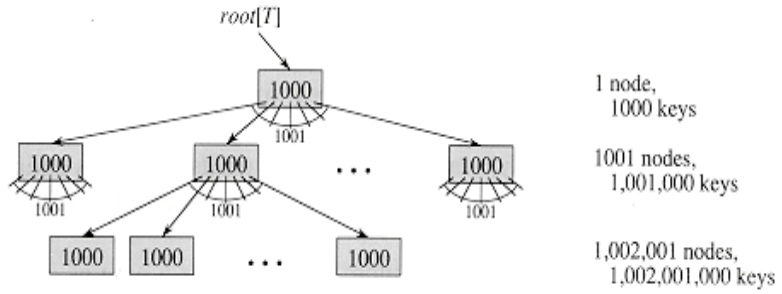


Figure 1.7: A B-tree of height 2 containing over one billion keys.

Each internal node and leaf contain 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is n[x], the number of keys in x.

The system can only keep a limited number of pages in main memory at any one time. We shall assume that pages no longer in use are flushed from main memory by the system; our B-tree algorithms will ignore this issue.

Since in most systems the running time of a B-tree algorithm is determined mainly by the number of DISK-READ and DISK-WRITE operations it performs, it is sensible to use these operations intensively by having them read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk page. The number of children a B-tree node can have is therefore limited by the size of a disk page.

For a large B-tree stored on a disk, branching factors between 50 and 2000 are often used, depending on the size of a key relative to the size of a page. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key. Figure 1.7 shows a B-tree with a branching factor of 1001 and height 2 that can store over one billion keys; nevertheless, since the root node can be kept permanently in main memory, only *two* disk accesses at most are required to find any key in this tree!

To keep things simple, we assume, as we have for binary search trees and red-black trees, that any "satellite information" associated

Space for learners:

with a key is stored in the same node as the key. In practice, one might actually store with each key just a pointer to another disk page containing the satellite information for that key. The pseudo code in this chapter implicitly assumes that the satellite information associated with a key, or the pointer to such satellite information, travels with the key whenever the key is moved from node to node. Another commonly used B-tree organization stores all the satellite information in the leaves and only stores keys and child pointers in the internal nodes, thus maximizing the branching factor of the internal nodes.

A ***B-tree*** T is a rooted tree (with root $root[T]$) having the following properties.

1. Every node x has the following fields:
   a. $n[x]$, the number of keys currently stored in node x,
   b. the $n[x]$ keys themselves, stored in non-decreasing order: $key_1[x] \le key_2[x] \le \ldots\ldots\ldots\ldots \le key_n[x][x]$, and
   c. $leaf[x]$, a Boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. If x is an internal node, it also contains $n[x] + 1$ pointers $c_1[x], c_2[x], \ldots, c_n[x]+1[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

   $$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \text{key}_{n[x]}[x] \le k_n[x]+1 .$$

4. Every leaf has the same depth, which is the tree's height h.
5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \le 2$ called the ***minimum degree*** of the B-tree:
   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

Space for learners:

b. Every node can contain at most $2t$ - 1 keys. Therefore, an internal node can have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t$ - 1 key.

The simplest B-tree occurs when $t = 2$. Every internal node then has either2, 3 or 4 children and we have a ***2-3-4tree.*** In practice, however, much larger values of t are typically used.

1.4.4 (a,b) TREE

An (a,b) Tree is one kind of a balanced binary search tree. An (a,b)-tree has all of its leaves at the same depth, and all internal nodes except for the root have between a and b children, where a and b are integers such that

- $2 \leq a \leq (b+1)/2$
- Each internal node except the root has at least **a** children and at most **b** children.
- The root has at most **b** children.

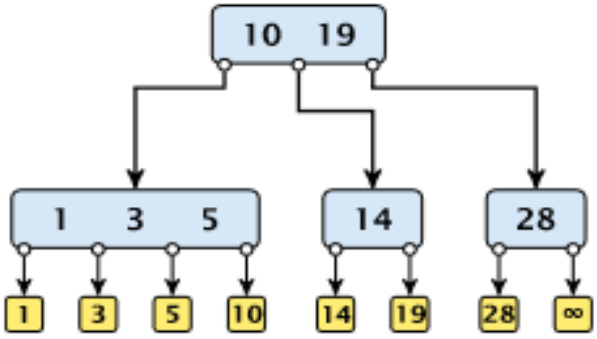


Figure 1.8: An Example of a (a,b) Tree

The insertion and deletion of (a,b) tree can be explained as follows-

a. INSERTION

The insertion can be start by adding into the proper leaf node. If the addition causes an overflow (b items), then split and propagate the middle item.

Space for learners:

Example: to insert 10 into the tree shown below-

Space for learners:

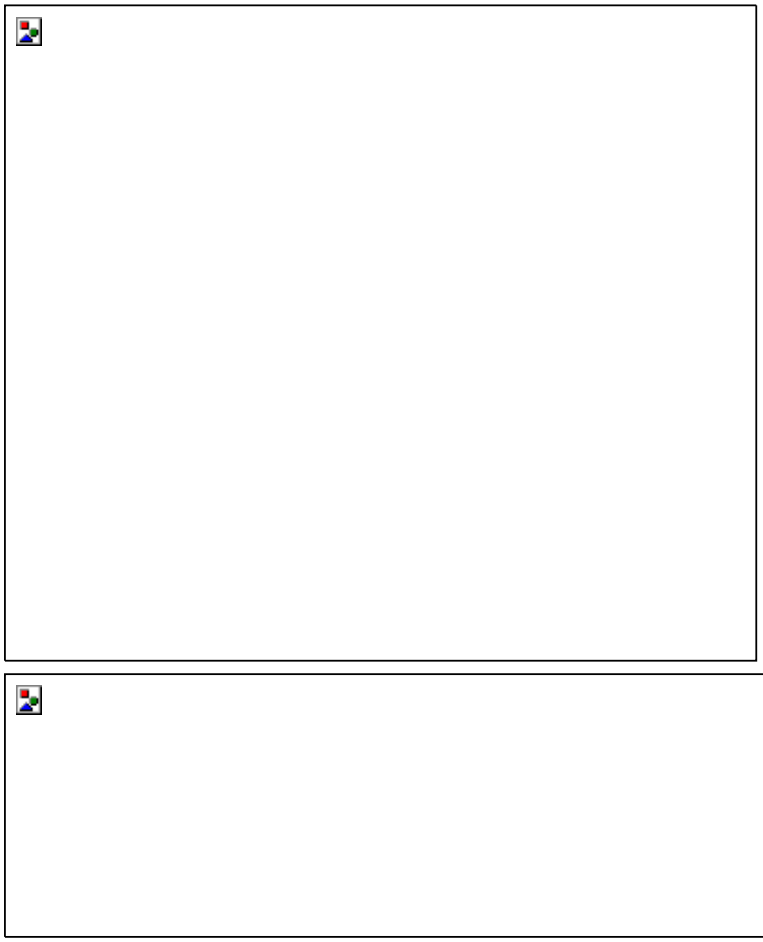


Figure 1.9:An example of insertion into a (a,b) Tree

b. DELETION

First, if the item you are deleting is not in a leaf node, then bring its predecessor up into its space and delete the predecessor item from its leaf node. This may require transfers and fusions. Fusions may cause underflow at the parent so in general this process has to be repeated up the tree.

A simple deletion example, requiring transfers only:

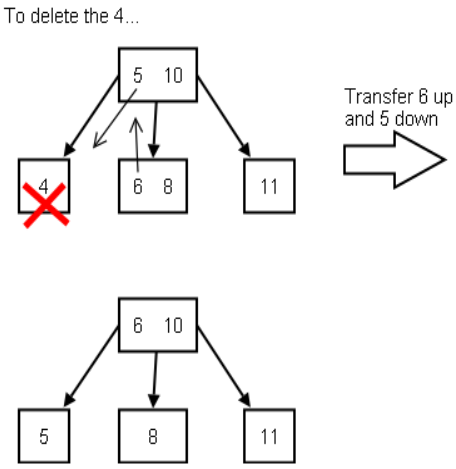


Figure 1.10: An example of deletion from a (a,b) Tree

1.4.5 RED BLACK TREE

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.

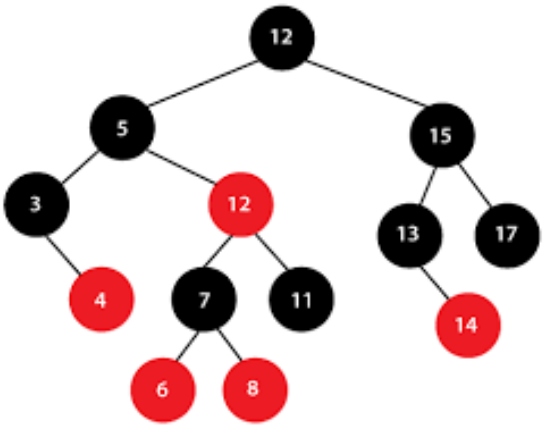


Figure 1.11: Red Black Tree

Space for learners:

It must be noted that as each node requires only 1 bit of space to store the colour information, these types of trees show identical memory footprint to the classic (uncoloured) binary search tree. Every red black tree has to follow the following rules-

1. Every node has a colour either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendant's NULL nodes has the same number of black nodes.

The time complexity of different operations in red-black tree is as follows-

Sl. No	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Most of the BST operations (e.g., search, max, min, insert, delete etc.) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

The detail operations on Red Black Tree are discussed in the next unit.

1.4.6 Splay Tree

A **splay tree** is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality. A good example is a network router. A network router receives network packets at a high rate from incoming connections

Space for learners:

and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. Splay trees can provide good performance in this situation.

The Splay is discussed in detailed in Unit 3.

Space for learners:

1.4.7 Ternary Search Tree

Ternary search trees are specialized structures for storing and retrieving strings. Like a binary search tree, each node holds a reference to the smaller and larger values. However, unlike a binary search tree, a ternary search tree doesn't hold the entire value in each node. Instead, a node holds a single letter from a word, and another reference—hence ternary—to a sub-tree of nodes containing any letters that follow it in the word.

Take a look at an example of a *Ternary Search Tree*, storing the following words: [“an”, “and”, “anti”, “end”, “so”, “top”, “tor”].Only filled (red) nodes are "key nodes", those correspond to words stored in the tree, while empty (white) vertices are just internal nodes.

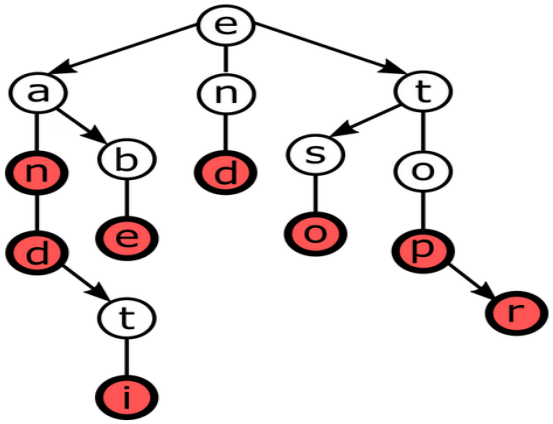


Figure 1.12: A Ternary Search Tree

Similarly, to tries, nodes in a *Ternary Search Tree* also need to store a Boolean value, to mark key nodes. The first difference that you can spot, with respect to a trie, is that a *Ternary Search Tree* stores characters in nodes, not in edges.As a matter of fact, each *Ternary Search Tree* node stores exactly three edges: to left, right, and middle children.

The "ternary search" part of the name should ring a bell, right? Indeed, *Ternary Search Trees* work somehow similarly to *BSTs*, only with three links instead of two. This is because they associate a character to each node, and while traversing the tree, we will choose which branch to explore based on how the next character in the input string compares to the current node's char.

Similarly to *BSTs*, in *Ternary Search Trees* the three outgoing edges of a node N partition the keys in its sub-tree; if N, holds character c, and its prefix in the tree (the *middle-node-path* from the *Ternary Search Tree*'s root to N, as we'll see) is the string s, then the following invariants hold:

1. All keys sL stored in the left sub-tree of N starts with s, are longer (in terms of number of characters) than s, and lexicographically less than s+c: sL<s+c (or, to put it in another way, the next character in sL is lexicographically less than c: if |s|=m,sL[m] < c).
2. All keys sR stored in the right sub-tree of N starts with s, are longer than s, and lexicographically greater than s+c: sR>s+c.
3. All keys in the middle sub-tree of N start with s+c.

This is best illustrated with an example: check out the graphic above and try to work out, for each node, the sets of sub-strings that can be stored in its 3 branches.

For instance, let's take the root of the tree:

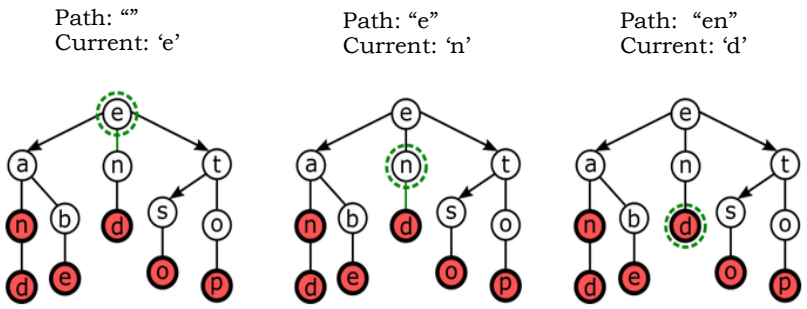
- Root's middle branch contains all keys starting with 'e';
- The left branch contains all keys whose first character precedes 'e' in lexicographic ordering: so, considering only lower-case letters in the English alphabet, one of 'a', 'b', 'c', 'd';
- Finally the right branch, which contains all keys that starts with letters from 'f' to 'z'.

Space for learners:

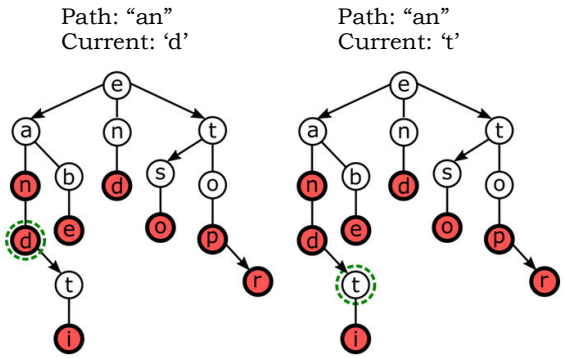
When we traverse a *Ternary Search Tree*, we keep track of a *"search string"*, as we do with tries: for each node N, it's the string that could be stored in N, and it's determined by the path from the root to N. The way we build this search string is, however, very different with respect to tries.

As you can see from the example above, a peculiarity of *Ternary Search Trees* is that a node's children have different meanings/functions.

The middle child is the one followed on characters match. It links a node N, whose path from root forms the string s, to a sub-tree containing all the stored keys that starts with s. When following the middle node we move one character forward in the search string.



The left and right child of a node, instead, doesn't let us advance in our path. If we had found i characters in a path from the root to N (i.e. we followed i middle links during traversal from root to N), and we traverse a left or right link, the current search string remains of length i.



Above, you can see an example of what happens when we follow a right-link. Differently from middle-links, we can't update the search

Space for learners:

string, so if on the left half current node corresponded to the word "and", on the right half the highlighted node, whose character is 't', corresponds to "ant": notice that there is no trace of traversing the previous node, holding 'd' (as there is also no trace of the root node, and it's like we didn't go through it, because our path had traversed a left-link from root to get to current node).

Left and right links, in other words, correspond to branching points after a common prefix: for "ant" and "and", for instance, after the first two characters (that can be stored only once, in the same path) we will need to branch out, to store both alternatives.

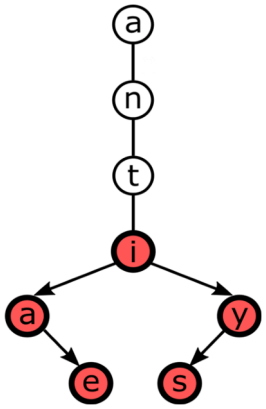
Which one gets the middle-link, and which one the left or right link? This is not determined beforehand, it only depends on the order they are inserted: first come, first serve! In the figure above, "and" was apparently stored before "anti".

Although Ternary Search Tree is an alternative to tries, but the question arises that how efficient this data structure than tries? So we will discuss and space and time complexity of Ternary Search Tree to get a better understanding of the concept.

Space Complexity

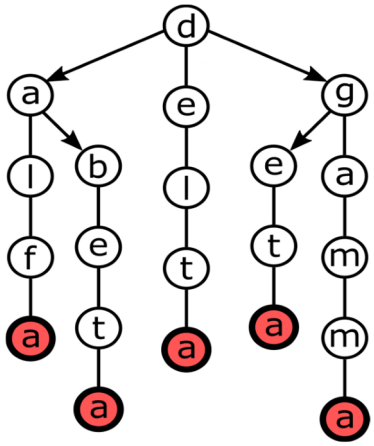
So, the question now arises: how many links (and nodes) are created for such *Ternary Search Tree*? To answer that, suppose we want to store n keys whose average length is w, then we can say that:

1. The minimum number of links needed we'll be w + n - 2: this is when all words share a prefix of length w-1, we have a middle-node path of |w-2| characters (and |w-1| nodes) from the root, and then we'll branch out n times at the very last character (with exactly 1 middle link, plus n-1 left/right links). An example of this edge case is shown in the figure below, with n=5 and w=4.



Space for learners:

2. The worst case scenario happens when no two words share a common prefix, we need to branch out at the root, and then for each word we'll have w-2 middle-links, for a total of n*(w-1) links. This is shown in the figure below, with n=5 and w~=4.



All other operations can be derived from tries in the same way, and can be implemented starting with a successful/unsuccessful search, or slightly modifying search.

Time Complexity

Performance-wise, a search hit or a search miss need, in the worst case, to traverse the longest path from the root to a leaf (there is no backtracking, so the longest path is a reliable measure of the cost of the worst case). That means search can perform at worst |A| * m characters comparisons (for completely skewed trees), where |A| is the size of the alphabet and m is the length of the searched string. Being the alphabet's size a constant, we can consider the time required for a successful search to be O(m) for a string of length m, and only differs for a constant factor from the trie's homologous.It is also provable that, for a balanced *Ternary Search Tree* storing n keys, a search miss requires O(log n) character comparisons at most (which is relevant for large alphabets, if |A| * m > n).

For remove: it can be performed as a successful search followed by some maintenance (performed during backtracking, it doesn't affect asymptotic analysis), and so its running time is also O(m) in the best case scenario, and an amortized O(log(n)) for unsuccessful removal.

Space for learners:

Finally add: it's also a search (either successful or unsuccessful) followed by the creation of a node chain with at most m nodes. Its running time is, then, also O(|A|*m).

Conclusions:

Ternary Search Trees are a valid alternative to *tries*, trading a slightly worse constant in their running time with an effective saving in the memory used.

Both adhere to the same interface, and allowed to implement efficiently some interesting searches on sets of strings.

The way *Ternary Search Trees* are implemented, however, allow for a trade-off between *memory* (which can be considerably less than the one needed for a trie storing the same set of strings) and *speed*, where both data structures have the asymptotic behavior, but *TSTs* are *a constant factor slower* than tries.

Space for learners:

1.5 BALANCING OF SEARCH TREES

Binary search tree is a best-suited data structure for data storage and retrieval when entire tree could be accommodated in the primary memory. However, this is true only when the tree is height-balanced. Lesser the height faster the search will be. Despite of the wide popularity of Binary search trees there has been a major concern to maintain the tree in proper shape. In worst case, a binary search tree may reduce to a linear link list, thereby reducing search to be sequential. Unfortunately, structure of the tree depends on nature of input. If input keys are not in random order the tree will become higher and higher on one side. In addition to that, the tree may become unbalanced after a series of operations like insertions and deletions. To maintain the tree in optimal shape many algorithms have been presented over the years. Most of the algorithms are static in nature as they take a whole binary search tree as input to create a balanced version of the tree. In this paper, few techniques have been discussed and analyzed in terms of time and space requirement.

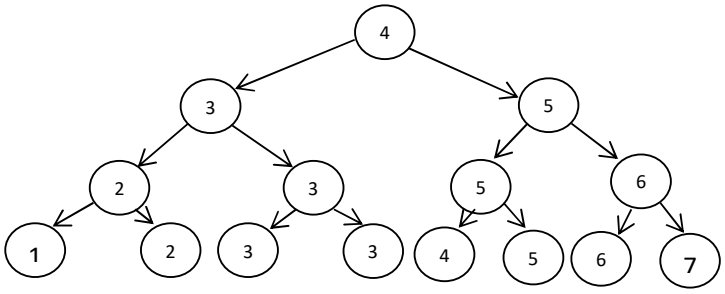


Figure 1.13: Binary Search Tree

Binary search tree is most basic, nonlinear data structure in computer science that can be defined as "a finite set of nodes that is either empty or consists of a root and two disjoint subsets called left and right sub-trees. Binary trees are most widely used to implement binary search algorithm for the faster data access. When memory allocation is static and data size is reasonably small, an array may be used instead to accomplish the same task. However, for large data set array is not a good option since it requires contiguous memory that system may not provide sometimes. In ideal situation, we would expect the tree to be of minimal height that is possible only when the tree is height balanced. With a n node random binary search tree search time grows only logarithmically O(lg(n)) as size of input grows. A binary search tree requires approximately 1.36(lg (n)) comparisons if keys are inserted in random order. It is also a well-known fact that total path length of a random tree can be further reduced by 27.85 percent by applying some rebalancing mechanism. There are two methods to rebalance a binary tree, dynamic rebalancing, and global (static) rebalancing. Both methods have advantages and disadvantages. Dynamic methods to create a balance binary search tree have been around since Adel'son-Velskii& Landis proposed the AVL Tree. Dynamic rebalancing methods maintain a tree in optimal shape by adjusting the tree whenever a node is inserted or deleted. Examples of this approach are height-balance tree, weight-balance tree, and B-trees. Rather than readjusting the tree every now and then global or static rebalancing methods allows the tree to grow unconstrained, and readjustment is done only when such a need is arises. To achieve this task many computer scientist have proposed various solutions.

Space for learners:

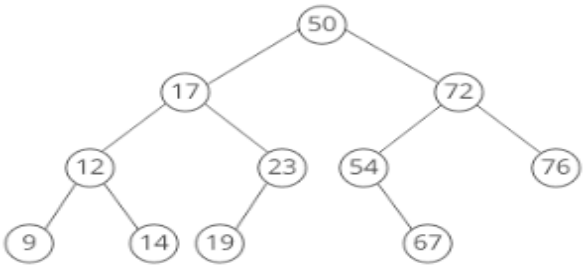


Figure 1.14: Balanced Binary Search Tree

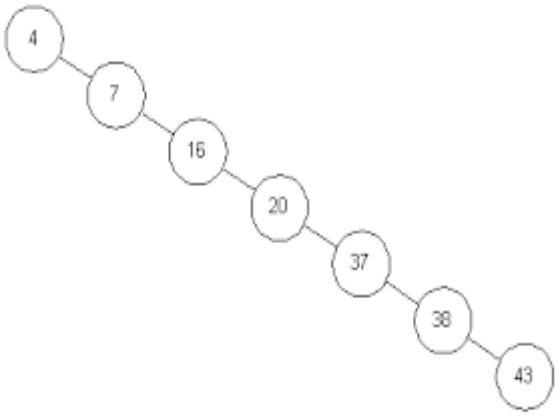


Figure 1.15 Unbalanced Binary Search Tree

Space for learners:

CHECK YOUR PROGRESS

1. Multiple Choice Questions:

(i) Which of the following is false about a binary search tree?

a) The left child is always lesser than its parent

b) The right child is always greater than its parent

c) The left and right sub-trees should also be binary search trees

d) In order sequence gives decreasing order of elements

(ii) What is the specialty about the in-order traversal of a binary search tree?

a) It traverses in a non-increasing order

b) It traverses in an increasing order

c) It traverses in a random fashion

d) It traverses based on priority of the node

(iii) What are the worst case and average case complexities of a binary search tree?

a) O(n), O(n)

b) O(logn), O(logn)

c) O(logn), O(n)

d) O(n), O(logn)

(iv) The minimum height of an AVL Tree with n node is

a) Ceil($\log_2(n+1)$)

b) 1.44 $\log_2 n$

c) Floor($\log_2(n+1)$)

d) 1.64 $\log_2 n$

(v) Which of the following is the most widely used external memory data structure?

a) AVL tree

b) B-tree

c) Red-black tree

d) Both AVL tree and Red-black tree.

(vi) What is the special property of red-black trees and what root should always be?

a) a color which is either red or black and root should always be black color only

b) height of the tree

c) pointer to next node

d) a color which is either green or black

Space for learners:

(vii) Which of the following is an application of Red-black trees?

a) used to store strings efficiently
b) used to store integers efficiently
c) can be used in process schedulers, maps, sets
d) for efficient sorting

(viii) When it would be optimal to prefer Red-black trees over AVL trees?

a) when there are more insertions or deletions
b) when more search is needed
c) when tree must be balanced
d) when log(nodes) time complexity is needed

(ix) Which of the following property of splay tree is correct?

a) it holds probability usage of the respective sub trees
b) any sequence of j operations starting from an empty tree with h nodes atmost, takes O(jlogh) time complexity
c) sequence of operations with h nodes can take O(logh) time complexity
d) splay trees are unstable trees

(x) How many child nodes does each node of Ternary Tree contain?

a) 4
b) 6
c) 5
d) 3

Space for learners:

2. Fill in the following blanks:

(i) To arrange a binary search tree in ascending order, we need_____ order traversal only.

(ii) Consider the binary search tree with n elements. The time required to search given element is_____.

(iii) In _____balance factor of a node is the difference between left sub-tree and right sub-tree.

(iv) The maximum height of an AVL tree with p nodes is_____.

(v) B-tree of order n is an order-n multi-way tree in which each non-root node contains _____keys.

(vi) A B-tree of order 4 and of height 3 will have a maximum of _____ keys.

(vii) The number of black nodes from the root to a node is the node's ____ ; the uniform number of black nodes in all paths from root to the leaves is called the ____ of the red–black tree.

(viii) In a Red-Black Tree, if a node is red, its child must be _____.

(ix) Self-adjusting binary search tree is called_____.

(x) Each node of ternary tree contains _____ number of child nodes.

Space for learners:

1.6 SUMMING UP

- A search tree is a tree data structure used for locating specific keys from within a set of elements. In order for a tree to function as a search tree, the key for each node must be greater

than any keys in sub-trees on the left, and less than any keys in sub-trees on the right.

- BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

  Left sub-tree (keys) < node (key) ≤ Right sub-tree (keys)

- AVL tree is a binary search tree in which the difference of heights of left and right sub-trees of any node is less than or equal to one.
- **B Tree** is a self-balancing data structure based on a specific set of rules for searching, inserting, and deleting the data in a faster and memory efficient way.
- An (a,b) Tree is one kind of a balanced binary search tree.
- A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black).
- A splay tree is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests.
- Ternary search trees are specialized structures for storing and retrieving strings. Like a binary search tree, each node holds a reference to the smaller and larger values. However, unlike a binary search tree, a ternary search tree doesn't hold the entire value in each node. Instead, a node holds a single letter from a word, and another reference—hence ternary—to a sub-tree of nodes containing any letters that follow it in the word.
- Balancing the tree makes for better search times O(**log**(n)) as opposed to O(n). As we know that most of the operations on Search Trees proportional to height of the Tree, So it is desirable to keep height small. It ensures that search time strict to O(log(n)) of complexity.

Space for learners:

1.7 ANSWERS TO CHECK YOUR PROGRESS

1.

(i) d), (ii) b), (iii) d), (iv) c), (v) b) , (vi) a), (vii) c), (viii) a), (ix) b), (x) d)

2.

(i) In, (ii) θ(log n), (iii) AVL Tree, (iv) log(p), (v) at least (n – 1)/2, (vi) 265, (vii) black depth, black height, (viii) Black, (ix) Splay Tree, (x) 3

1.8 POSSIBLE QUESTIONS

Short answer type questions:

1. What is a search tree?
2. What is the maximum number of nodes in a complete binary search tree?
3. How to search for a key in a binary search tree? (Write the procedural code).
4. In a binary search tree traversal, in which order traversal we get the elements in Ascending order?
5. What is an AVL Tree?
6. In an AVL tree how many types of rotation is possible?
7. What kind of balancing is done on AVL tree?
8. Which tree data structure is most widely used in external memory data structure?
9. What is the best case height of B-tree of order n and which has k nodes?
10. What is (a,b) Tree data structure?
11. Under what criteria (a,b) Tree is called a B Tree?
12. What is the Red Black tree data structure?
13. What is Splay Tree?
14. Why do you prefer Splay Tree data structure?

Space for learners:

15. What are the disadvantages of Splay tree?

Long answer type questions:

1. Explain the difference between Binary Tree and Binary Search Tree with an example.
2. (a) Draw a binary search tree that is created if the following numbers are inserted in the tree in the given order.

   12 15 3 35 21 42 14

   (b) Draw a balanced binary search tree containing the same numbers given in part (a).
3. Define B-tree. Explain in detail about the insertion and deletion operations in B-tree.
4. What is a Splay Tree? Explain the operations to be performed on a splay tree with diagram.
5. Draw a clearly labeled suffix tree for the string addaadd#.

Space for learners:

1.9 REFERENCES AND SUGGESTED READINGS

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *"Introduction to Algorithms"*, The MIT Press.
2. Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and AlgorithmicPuzzles*5th ed. Edition, The Career Monk.

---×---

UNIT 2: AVL TREES AND RED-BLACK TREES

Space for learners:

Unit Structure:

2.1 INTRODUCTION

In this unit you will be able to learn advanced search tree viz. AVL Tree and Red-Black Tree. AVL tree is a binary search tree in which difference of heights of left and right sub-trees of any node is less than or equal to 1. The technique of balancing height of binary trees was developed by Georgy Adelson-Velskii, and Evgenii Landis hence it is called AVL (after the name of its inventors). Red-Black tree is another member of binary search tree family. Like AVL tree,

a red-black tree has also self-balancing properties. The structure of a red black tree follows certain rules to ensure that the tree is always balanced.

Space for learners:

2.2 UNIT OBJECTIVES

After going to this unit, you will be able to:

- *understand* the fundamental concepts of AVL and Red Black Tree.
- *know* why balancing is important in a binary search tree
- *understand* types of rotation can performed in an AVL tree.
- *get an idea* when to perform single rotation and when to perform double rotation in AVL Tree.
- *know* how insertion and deletion can be performed in AVL Tree.
- *understand*the properties of the Red Black Tree.
- *learn*about Recolor, Rotation, Rotation followed by Recolor.
- *know* how insertion and deletion can be performed in Red Black Tree.

2.3 AVL TREE

2.3.1 Introduction

AVL tree is a binary search tree in which the difference of heights of left and right sub-trees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by two Soviet scientists Georgy Adelson-Velskii, and Evgenii Landis and hence given the short form as AVL tree or Balanced Binary Tree. The AVL was published in 1962 in a paper *"An algorithm for the organization of information"* in the proceedings of USSR Academy of Sciences (in Russian), which was later translated to English by Myron J. Ricci in *Soviet Mathematics – Doklady.*

An AVL tree can be defined as follows:

Let T be a non-empty binary tree with T_L and T_R as its left and right sub-trees. The tree is height balanced if:

- T_L and T_R are height balanced
- h_L - h_R<= 1, where h_L - h_R are the heights of T_L and T_R

The Balance factor of a node in a binary tree can have value 1, -1, 0, depending on whether the height of its left sub-tree is greater, less than or equal to the height of the right sub-tree.

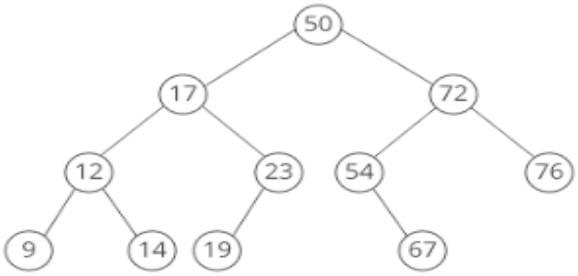


Figure2.1: Balanced Binary Search Tree

In the above figure the value of BF is 1 hence this is a balance binary tree.

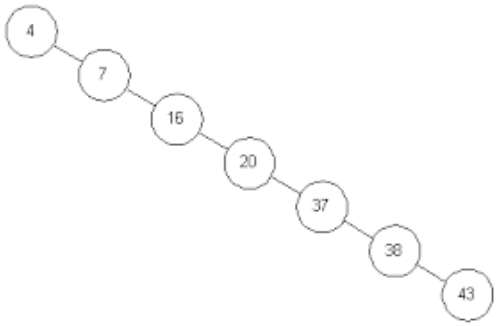


Figure2.2: Unbalanced Binary Search Tree

In the Figure 2.2 the value of BF is less than -1 hence this is an unbalance binary tree.

Operations on Binary Search Trees like search, insertion, deletion take O(h) time, where 'h' represents the height of the tree. In case of a Binary Search Trees in which all the nodes have only one child node or no child node (also called skewed Binary Search Tree) as shown in the Figure 2.2, performing these operation takes O(n) time, where 'n' denotes the number of nodes in the tree.

Space for learners:

To tackle this inefficiency, the tree needs to reconstructed after every operation in such a way that it always maintains logarithmic height, thereby reducing the time complexity for all operations toO(log n). This means that the height of the tree must be maintained after every insertion and deletion such that the time complexity for every operation performed remains O(h).
The C code structure for AVL tree is-

```
typedefstruct AVL
{
        int data;
        struct AVL *lnode;
        struct AVL *rnode;
        int height;
}AVL;
```

2.3.2 AVL Tree Rotation

In AVL tree, after performing operations like insertion and deletion we need to check the balance factor (BF) of every node in the tree. If every node satisfies the balance factor condition, then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation, we use rotation operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

Rotation is the process of moving nodes either to left or to right to make the tree balanced.

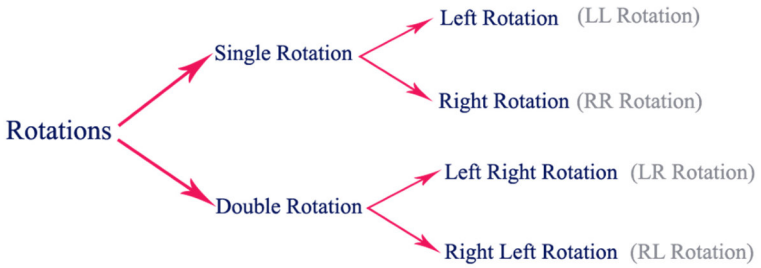


Figure 2.3: Types of rotation of AVL Tree

There are four rotations and they are classified into two types.

Space for learners:

- **Single Rotation**

(i) Left Rotation (LL Rotation)

If a tree becomes unbalanced, when a node is inserted into the right sub-tree of the right sub-tree, then we perform a single left rotation. Look at the example given below-

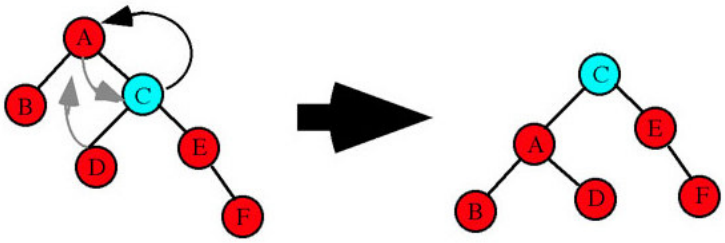


Figure2.4: Left Rotation of AVL Tree

From the Figure 2.4 (a) it is clear that the tree in unbalance and a rotation is required to balance the tree. In this case Left rotation will be applied as it is unbalanced in right sub-tree. Node C is moved up. Node A becomes the left child of Node C and Node D becomes the right child of A

(ii) Right Rotation (LR Rotation)

AVL tree may become unbalanced, if a node is inserted in the left sub-tree of the left sub-tree. The tree then needs a right rotation. An example is given below-

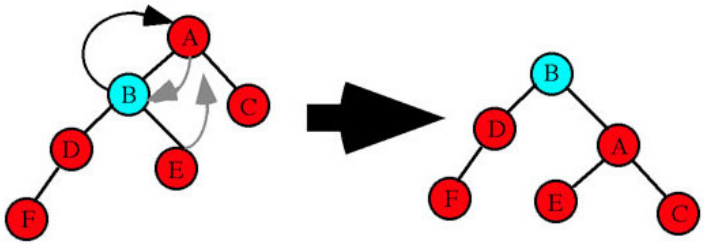


Figure 2.5: Right Rotation of AVL Tree

From the Figure 2.5 (a), it is evident that the tree is become unbalanced in the left sub-tree so right rotation is required to balance the tree. Here Node B is moved up. Node A becomes the right child of Node B and Node E becomes the left child of A. Study carefully the arrangement of the keys in a binary tree and you will

Space for learners:

see that this new balanced arrangement also maintains the binary tree quality, i.e. all nodes in the left sub-tree have keys less than a given node and all nodes in the right sub-tree have keys greater than a given node.

Space for learners:

- **Double Rotation**

(i) Left Right Rotation (LR Rotation)

Double rotations are slightly complex version of already explained versions of rotations. A double rotation, in which, a *left rotation* is followed by a *right rotation*.

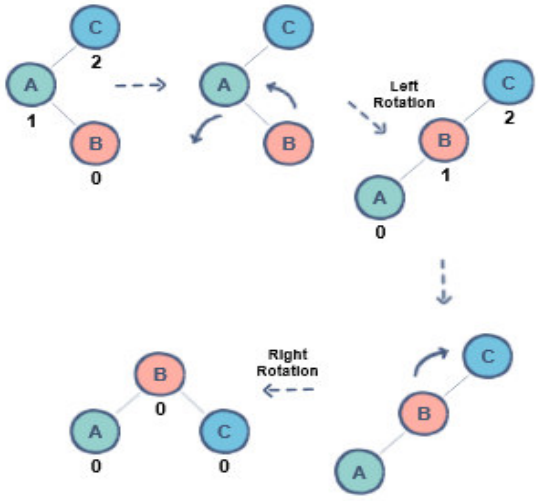


Figure 2.6: Left Right Rotation of AVL Tree

In the Figure 2.6, **node B** is causing an imbalance resulting in **node C** to have a balance factor of 2. As **node B** is inserted in the right sub-tree of **node A**, a *left rotation* needs to be applied. However, a single rotation will not give us the required results. Now, all we have to do is apply the *right rotation* as shown before to achieve a balanced tree.

(ii) Right Left Rotation (RL Rotation)

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

Space for learners:

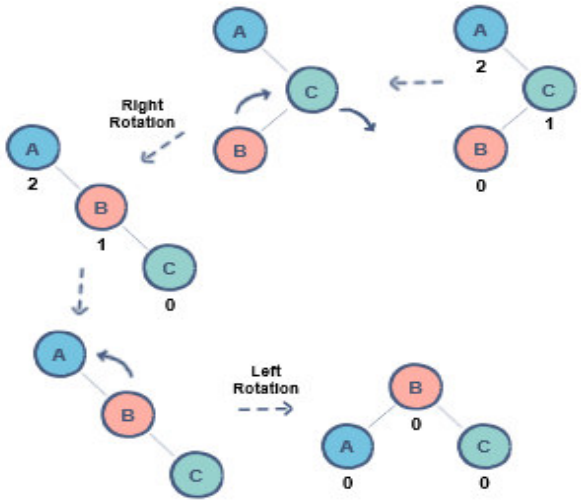


Figure 2.7: Right Left Rotation of AVL Tree

In the Figure 2.7, node B is causing an imbalance resulting in node A to have a balance factor of 2. As node B is inserted in the left sub-tree of node C, a *right rotation* needs to be applied. However, just as before, a single rotation will not give us the required results. Now, by applying the *left rotation* as shown before, we can achieve a balanced tree.

2.3.3 Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree.

We use the following steps to search an element in AVL tree...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left sub-tree.

Step 6 - If search element is larger, then continue the search process in right sub-tree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.

Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

The C code for traversal of AVL is given below-

Pre Oder Traversal

```
void preorder(node *root)
{
        if(root!=NULL)
        {
        printf("%d",root->info);
        preorder(root->lnode);
        preorder(root->rnode);
        }
}
```

In Oder Traversal

```
voidinorder(AVL *root)
{
        if(root!=NULL)
        {
        inorder(root->lnode);
        printf("%d",root->data);
        inorder(root->rnode);
        }
}
```

Space for learners:

2.3.4 Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2 - After insertion, check the **Balance Factor** of every node.

Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

2.3.5 Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. There, the effective deletion of the subject node or the replacement node decreases the height of the corresponding child tree either from 1 to 0 or from 2 to 1, if that node had a child.

Starting at this sub-tree, it is necessary to check each of the ancestors for consistency with the invariants of AVL trees. This is called "retracing".

Since with a single deletion the height of an AVL sub-tree cannot decrease by more than one, the temporary balance factor of a node will be in the range from −2 to +2. If the balance factor remains in the range from −1 to +1 it can be adjusted in accord with the AVL rules. If it becomes ±2 then the sub-tree is unbalanced and needs to be rotated. (Unlike insertion where a rotation always balances the tree, after delete, there may be BF(Z) ≠ 0 (see figures 2 and 3), so that after the appropriate single or double rotation the height of the rebalanced sub-tree decreases by one meaning that the tree has to be rebalanced again on the next higher level.) The height of the sub-tree rooted by N has decreased by 1. It is already in AVL shape.

Space for learners:

2.3.6 Complete Program of AVL Tree

Space for learners:

The complete C program for AVL tree is demonstrated below (the program in tested in Linux Environment)-

```
#include<stdio.h>
#include<stdlib.h>

typedefstruct AVL
{
        int info;
        struct AVL *lnode,*rnode;
        intht;
}AVL;

AVL *insert(AVL *,int);
AVL *Delete( AVL *,int);
void preorder(AVL *);
voidinorder(AVL *);
int height(AVL *);
AVL *rotateright(AVL *);
AVL *rotateleft(AVL *);
AVL *RR(AVL *);
AVL *LL(AVL *);
AVL *LR(AVL *);
AVL *RL(AVL *);
int BF(AVL *);

int main()
{
        AVL *root=NULL;
        intx,n,i,ch;
        do
        {
                printf("\n1)Create:");
                printf("\n2)Insert:");
                printf("\n3)Delete:");
                printf("\n4)Print:");
                printf("\n5)Exit:");
                printf("\n\nEnter Your Choice:");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:
                                printf("\nEnter no. of elements:");
                                scanf("%d",&n);
                                printf("\nEnter tree data:");
```

```
                        root=NULL;
                        for(i=0;i<n;i++)
                        {
                                scanf("%d",&x);
                                root=insert(root,x);
                        }
                        break;
                case 2:
                        printf("\nEnter a data:");
                        scanf("%d",&x);
                        root=insert(root,x);
                        break;
                case 3:
                        printf("\nEnter a data:");
                        scanf("%d",&x);
                        root=Delete(root,x);
                        break;
                case 4:
                        printf("\nPreorder sequence:\n");
                        preorder(root);
                        printf("\n\nInorder sequence:\n");
                        inorder(root);
                        printf("\n");
                        break;
                }
        }while(ch!=5);
        return 0;
}

AVL * insert(AVL *root,int x)
{
        if(root==NULL)
        {
                root=(AVL*)malloc(sizeof(AVL));
                root->info=x;
                root->lnode=NULL;
                root->rnode=NULL;
        }
        else
                if(x > root->info) // insert in right node subtree
                {
                        root->right=insert(root->rnode,x);
                        if(BF(root)==-2)
                                if(x>root->rnode->info)
                                        root=RR(root);
                                else
                                        root=RL(root);
```

Space for learners:

```
                }
                else
                        if(x<root->info)
                        {
                                root->lnode=insert(root->lnode,x);
                                if(BF(root)==2)
                                        if(x < root->lnode->info)
                                                root=LL(root);
                                        else
                                                root=LR(root);
                        }
        root->ht=height(root);
        return(root);
}

AVL * Delete(AVL *root,int x)
{
        AVL *p;
        if(root==NULL)
        {
                return NULL;
        }
        else
                if(x > root->info) // insert in right subtree
                {
                        root->rnode=Delete(root->rnode,x);
                        if(BF(root)==2)
                                if(BF(root->lnode)>=0)
                                        root=LL(root);
                                else
                                        root=LR(root);
                }
                else
                        if(x<root->info)
                        {
                                root->lnode=Delete(root->lnode,x);
                                if(BF(root)==-2) //Rebalance during
                        windup
                                        if(BF(root->rnode)<=0)
                                                root=RR(root);
                                else
                                        root=RL(root);
                        }
                        else
                        {
                                //info to be deleted is found
                                if(root->rnode!=NULL)
```

Space for learners:

```
                        { //delete its inordersuccesor
                                p=root->rnode;
                                while(p->lnode!= NULL)
                                        p=p->lnode;
                                root->info=p->info;
                                root->rnode=Delete(root->rnode,p-
                        >info);
                                if(BF(root)==2)//Rebalance during
                        windup
                                        if(BF(root->lnode)>=0)
                                                root=LL(root);
                                        else
                                                root=LR(root);\
                                }
                                else
                                        return(root->lnode);
                        }
                        root->ht=height(root);
                        return(root);
}

int height(AVL *root)
{
        intlh,rh;
        if(root==NULL)
                return(0);
        if(root->lnode==NULL)
                lh=0;
        else
                lh=1+root->lnode->ht;
        if(root->rnode==NULL)
                rh=0;
        else
                rh=1+root->rnode->ht;
                if(lh>rh)
                        return(lh);
        return(rh);
}

AVL * rotateright(AVL *x)
{
        AVL *y;
        y=x->lnode;
        x->lnode=y->rnode;
        y->rnode=x;
        x->ht=height(x);
        y->ht=height(y);
```

Space for learners:

```
        return(y);
}

AVL * rotateleft(AVL *x)
{
        AVL *y;
        y=x->rnode;
        x->rnode=y->lnode;
        y->lnode=x;
        x->ht=height(x);
        y->ht=height(y);
        return(y);
}

AVL * RR(AVL *root)
{
        root=rotateleft(root);
        return(root);
}

AVL * LL(AVL *root)
{
        root=rotateright(root);
        return(root);
}

AVL * LR(AVL *root)
{
        root->lnode=rotateleft(root->lnode);
        root=rotateright(root);
        return(root);
}

AVL * RL(AVL *root)
{
        root->rnode=rotateright(root->rnode);
        root=rotateleft(root);
        return(root);
}

int BF(AVL *root)
{
        intlh,rh;
        if(root==NULL)
        return(0);
        if(root->lnode==NULL)
                lh=0;
```

Space for learners:

```
        else
                lh=1+root->lnode->ht;

        if(root->rnode==NULL)
                rh=0;
        else
                rh=1+root->rnode->ht;
        return(lh-rh);
}
void preorder(AVL *root)
{
        if(root!=NULL)
        {
                printf("%d(Bf=%d)",root->info,BF(root));
                preorder(root->lnode);
                preorder(root->rnode);
        }
}

voidinorder(AVL *root)
{
        if(root!=NULL)
        {
                inorder(root->lnode);
                printf("%d(Bf=%d)",root->info,BF(root));
                inorder(root->rnode);
        }
}
```

Space for learners:

2.4 RED BLACK TREE

2.4.1 Introduction

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.

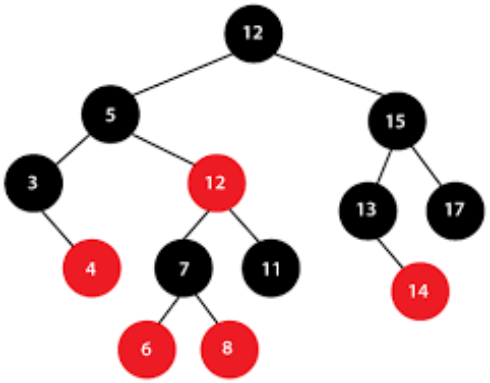


Figure 2.8: Red-Black Tree

It must be noted that as each node requires only 1 bit of space to store the colour information, these types of trees show identical memory footprint to the classic (uncoloured) binary search tree. Every red black tree has to follow the following rules-

1. Every node has a colour either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendant's NULL nodes has the same number of black nodes.

Time complexity of red-black tree is given below-

Sl. No	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Most of the BST operations (e.g., search, max, min, insert, delete etc.) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

Space for learners:

2.4.2 Insertion Operation in Red-Black Tree

Space for learners:

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

1. Recolor
2. Rotation
3. Rotation followed by Recolor

The insertion operation in Red Black tree is performed using the following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the newNodeas Root node with color Black and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

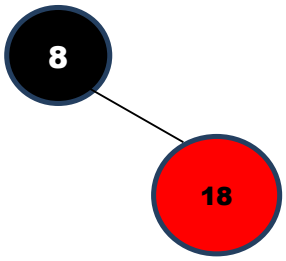
Let us create a RED BLACK tree by inserting following sequence of number 8,18,5,15,17,25,40,80.

Insert (8)

The tree is Empty. So insert new Node as Root node with black colour.

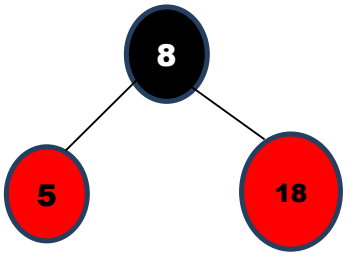
Insert (18)

The tree is not Empty. So inert new Node with red colour.



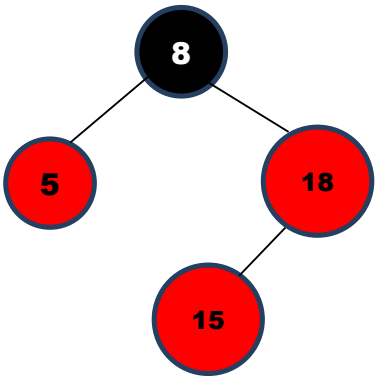
Insert (5)

The tree is not Empty. So inert new Node with red colour



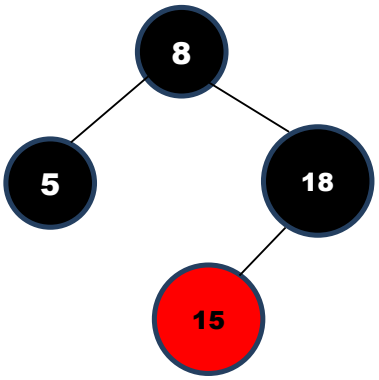
Insert (15)

The tree is not Empty. So inert new Node with red colour



In the above diagram two consecutive red nodes (18 and 15). The new Node's parent and sibling colour is red and parent's parent is root node. So, we have to user RECOLOR to make red black tree.

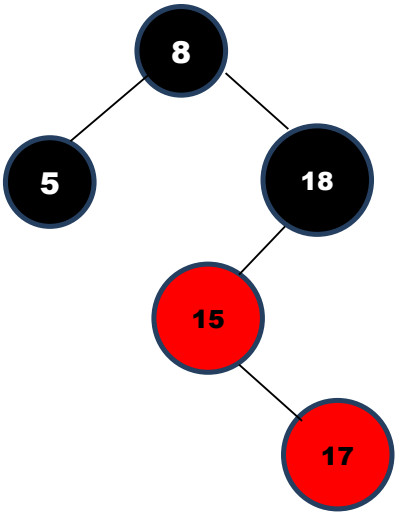
Space for learners:



After recolor operation, the tree is satisfying all Red Black Tree properties.

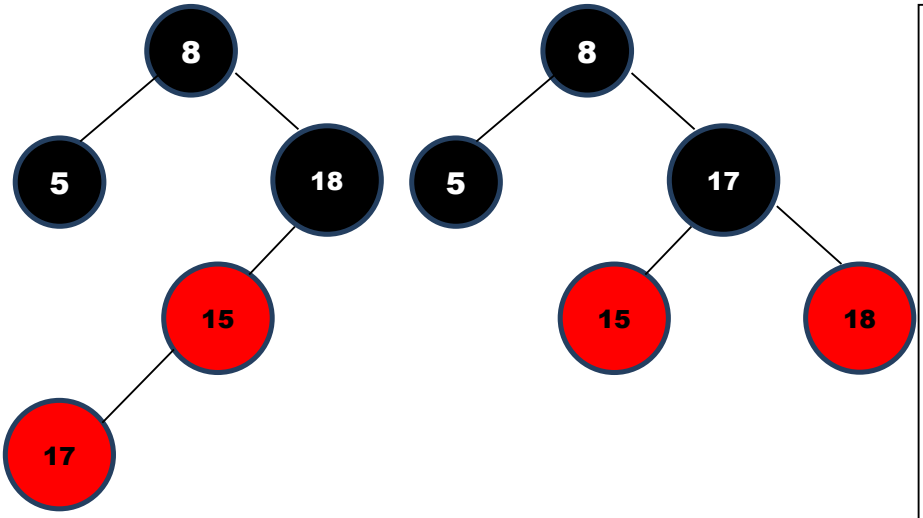
Insert (17)

The tree is not Empty. So inert new Node with red colour



In the above diagram two consecutive red nodes (15 and 17). The new Node's parent sibling is NULL. So, we need rotation. Here we need LR rotation and RECOLOR.

Space for learners:

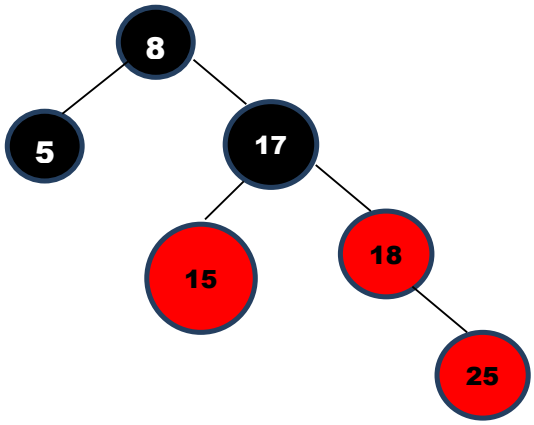


After Left Rotation and Recolour

After Right Rotation

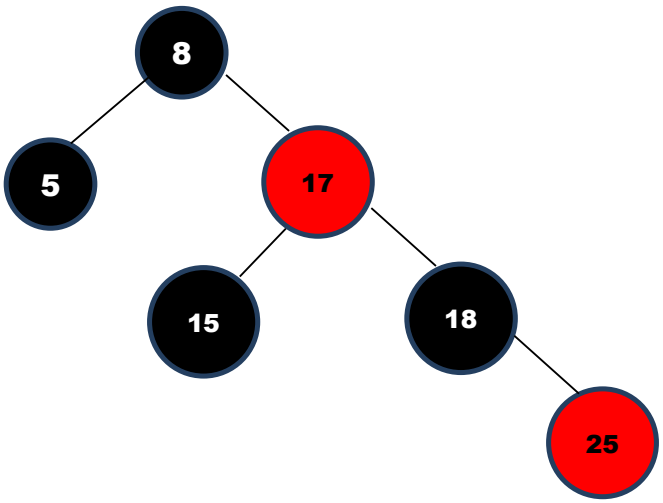
Insert (25)

The tree is not Empty. So inert new Node with red colour



In the above diagram two consecutive red nodes (18 and 25). The new Node's parent sibling colour is Red and parent's parent is not root node. So, we use RECOLOR and Recheck.

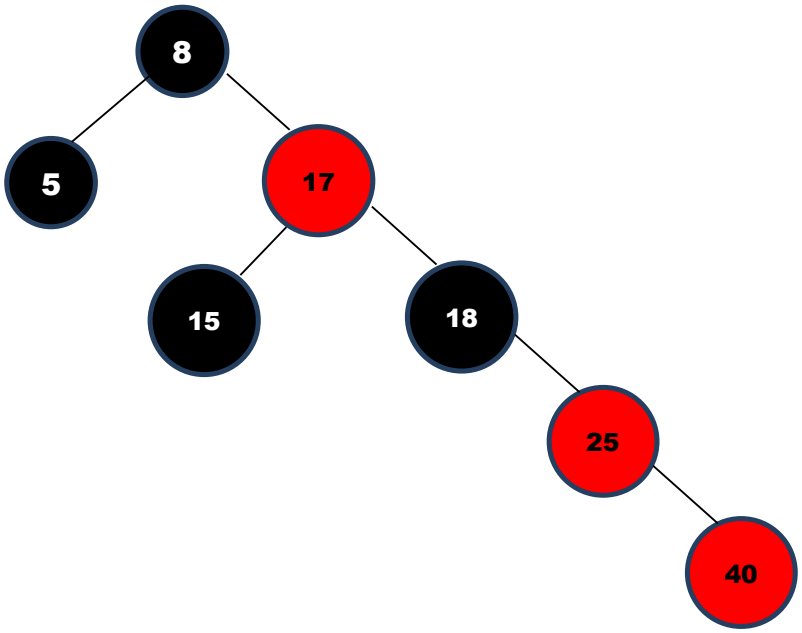
Space for learners:



After Recolour operation the tree is satisfying all Red Black Tree properties.

Insert (40)

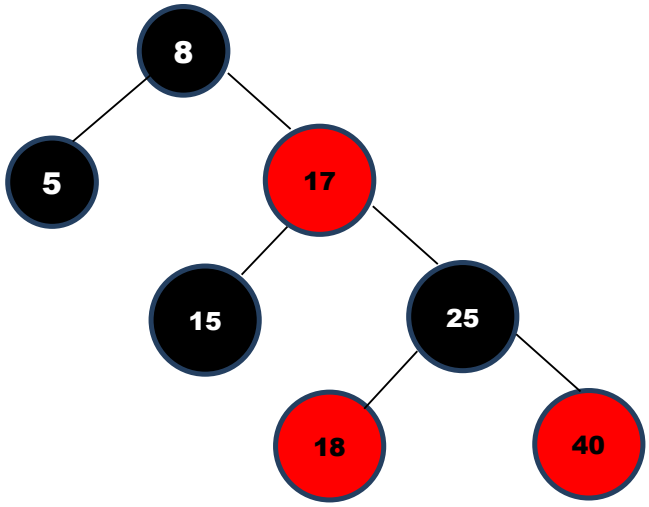
The tree is not Empty. So inert new Node with red colour



In the above diagram two consecutive red nodes (25 and 40). The newNode's parent sibling is NULL. So we need RECOLOR and Recheck. Here we use LL Rotation and Recheck.

Space for learners:

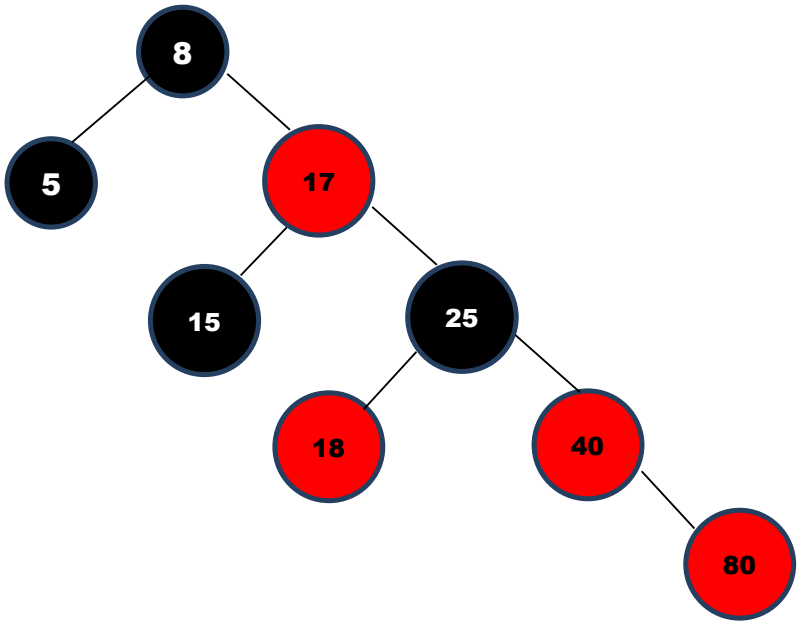
Space for learners:



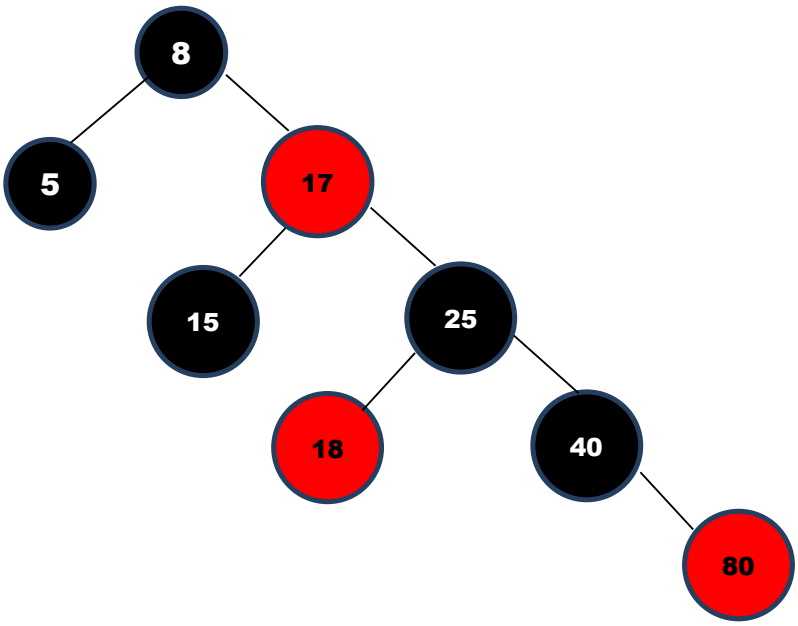
After LL Rotation and Recolour operation, the tree is satisfying all Red Black Tree properties.

Insert (80)

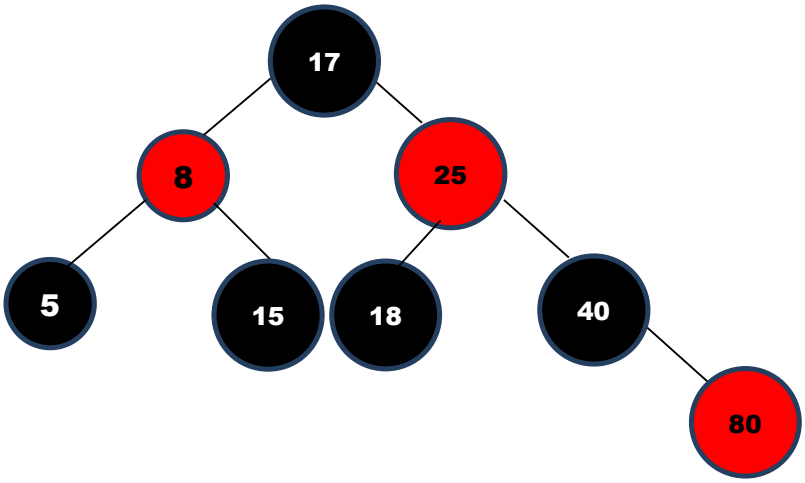
The tree is not Empty. So inert new Node with red colour



In the above diagram two consecutive red nodes (40 and 80). The new Node's parent sibling colour is Red and parent's parent node is not root node. So we need RECOLOR and Recheck.



After LL Rotation and Recolour operation, the tree is satisfying all Red Black Tree properties as shown below.



Finally, above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black Tree.

2.4.3 Deletion Operation in Red Black Tree

The deletion operation in Red-Black Tree is fairly a complex process. To understand deletion in Red Black Tree, notion of double black is used. When a black node is deleted and replaced by black

Space for learners:

child, the child is marked as *double black*. The main job now is to convert this *double black* to *single black.* The detail steps for deletion from a Red Black Tree is given below-

Space for learners:

1. **Perform standard Binary Search Tree Deletion**. When we perform standard deletion in BST, we end up deleting a node which is either a leaf or has only one child. So we need to handle cases where a node is leaf or has one child. Let *v* be the node to be deleted and *u* be the child that replaces *v*.
2. **Simple Case:** if either *u* or *v* is red, we mark the replaced child as black and there is no change in black height. Point to be noted that both *u* and *v* cannot be red as *v* is parent of *u* and two consecutive red nodes are not allowed in red black tree.

3. If both *u* and *v* are black

3.1 First we have to colour *u* as double black. Next our task is to reduce to convert this double black to single black. So the deletion of a black leaf also causes a double black.

3.2 Do the following, while the current node *u* is double black or it is not the root. Let sibling of node is *s.*

(a) **If sibling is black and at least one of sibling's children is red,** then have to perform rotation(s). Let the red child of *s* be *r*. This can be divided in sub cases depending upon positions of *s* and *r.*

(i) Left Left Case (*s*is the left child of its parent and *r* is the left child of *s* or both children of *s* are red).

(ii) Left Right Case (*s* is the left child of its parent and *r* is the right child of its parent.

(iii) Right Right Case (*s* is right child of its parent and *r* is right child of s or both children of s are red).

(iv) Right Left Case (*s* is right child of its parent and *r* is left child of *s*).

(b) **If sibling is black and it's both children are black**, performs recoloring, and recurs for the parent if

parent is black.In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

(c) **If sibling is red**, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black. This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two sub-cases.

(i) Left Case (*s* is left child of its parent). We right rotate the parent *p*.

(ii) Right Case (*s* is right child of its parent). We left rotate the parent *p*.

3.3 If *u* is root, make it single black and return (Black height of complete tree reduces by 1).

Space for learners:

2.4.4 Complete Program of Red Black Tree

/* Implementing Red-Black Tree in C*/

```
#include <stdio.h>
#include <stdlib.h>

enum nodeColor {
  RED,
  BLACK
};

Typedef struct tree {
int info, color;
struct tree *link[2];
}RBtree;

    RBtree *root = NULL;

/* Create a red-black tree */
RBtree *createNode(intval) {
RBtree *newnode;
newnode = (RBtree *)malloc(sizeof(RBtree));
newnode->info = val;
newnode->color = RED;
newnode->link[0] = newnode->link[1] = NULL;
```

```
returnnewnode;
}

/* Insert an node */
void insert(intval) {
RBtree *stack[98], *ptr, *newnode, *xPtr, *yPtr;
intdir[98], ht = 0, index;
ptr = root;
if (!root) {
root = createNode(val);
return;
  }

stack[ht] = root;
dir[ht++] = 0;
while (ptr != NULL) {
if (ptr->info == val) {
printf("Duplicates Not Allowed!!\n");
return;
    }
index = (info - ptr->info) > 0 ? 1 : 0;
stack[ht] = ptr;
ptr = ptr->link[index];
dir[ht++] = index;
  }
stack[ht - 1]->link[index] = newnode = createNode(val);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
if (dir[ht - 2] == 0) {
yPtr = stack[ht - 2]->link[1];
if (yPtr != NULL &&yPtr->color == RED) {
stack[ht - 2]->color = RED;
stack[ht - 1]->color = yPtr->color = BLACK;
ht = ht - 2;
      } else {
if (dir[ht - 1] == 0) {
yPtr = stack[ht - 1];
        } else {
xPtr = stack[ht - 1];
yPtr = xPtr->link[1];
xPtr->link[1] = yPtr->link[0];
yPtr->link[0] = xPtr;
stack[ht - 2]->link[0] = yPtr;
        }
xPtr = stack[ht - 2];
xPtr->color = RED;
yPtr->color = BLACK;
xPtr->link[0] = yPtr->link[1];
```

Space for learners:

```
yPtr->link[1] = xPtr;
if (xPtr == root) {
root = yPtr;
        } else {
stack[ht - 3]->link[dir[ht - 3]] = yPtr;
        }
break;
      }
    } else {
yPtr = stack[ht - 2]->link[0];
if ((yPtr != NULL) && (yPtr->color == RED)) {
stack[ht - 2]->color = RED;
stack[ht - 1]->color = yPtr->color = BLACK;
ht = ht - 2;
      } else {
if (dir[ht - 1] == 1) {
yPtr = stack[ht - 1];
        } else {
xPtr = stack[ht - 1];
yPtr = xPtr->link[0];
xPtr->link[0] = yPtr->link[1];
yPtr->link[1] = xPtr;
stack[ht - 2]->link[1] = yPtr;
        }
xPtr = stack[ht - 2];
yPtr->color = BLACK;
xPtr->color = RED;
xPtr->link[1] = yPtr->link[0];
yPtr->link[0] = xPtr;
if (xPtr == root) {
root = yPtr;
        } else {
stack[ht - 3]->link[dir[ht - 3]] = yPtr;
        }
break;
      }
    }
  }
root->color = BLACK;
}
```

/* Delete a node*/

```
void delete(intval) {
RBtree *stack[98], *ptr, *xPtr, *yPtr;
RBtree *pPtr, *qPtr, *rPtr;
intdir[98], ht = 0, diff, i;
```

Space for learners:

```
enumnodeColor color;

if (!root) {
printf("Tree not available\n");
return;
  }

ptr = root;
while (ptr != NULL) {
if ((val - ptr->info) == 0)
break;
diff = (val - ptr->info) > 0 ? 1 : 0;
stack[ht] = ptr;
dir[ht++] = diff;
ptr = ptr->link[diff];
  }

if (ptr->link[1] == NULL) {
if ((ptr == root) && (ptr->link[0] == NULL)) {
free(ptr);
root = NULL;
    } else if (ptr == root) {
root = ptr->link[0];
free(ptr);
    } else {
stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
    }
  } else {
xPtr = ptr->link[1];
if (xPtr->link[0] == NULL) {
xPtr->link[0] = ptr->link[0];
color = xPtr->color;
xPtr->color = ptr->color;
ptr->color = color;

if (ptr == root) {
root = xPtr;
      } else {
stack[ht - 1]->link[dir[ht - 1]] = xPtr;
      }

dir[ht] = 1;
stack[ht++] = xPtr;
    } else {
      i = ht++;
while (1) {
dir[ht] = 0;
```

Space for learners:

```
stack[ht++] = xPtr;
yPtr = xPtr->link[0];
if (!yPtr->link[0])
break;
xPtr = yPtr;
     }

dir[i] = 1;
stack[i] = yPtr;
if (i > 0)
stack[i - 1]->link[dir[i - 1]] = yPtr;

yPtr->link[0] = ptr->link[0];

xPtr->link[0] = yPtr->link[1];
yPtr->link[1] = ptr->link[1];

if (ptr == root) {
root = yPtr;
     }

color = yPtr->color;
yPtr->color = ptr->color;
ptr->color = color;
   }
  }

if (ht < 1)
return;

if (ptr->color == BLACK) {
while (1) {
pPtr = stack[ht - 1]->link[dir[ht - 1]];
if (pPtr&&pPtr->color == RED) {
pPtr->color = BLACK;
break;
     }

if (ht < 2)
break;

if (dir[ht - 2] == 0) {
rPtr = stack[ht - 1]->link[1];

if (!rPtr)
break;
```

Space for learners:

```
if (rPtr->color == RED) {
stack[ht - 1]->color = RED;
rPtr->color = BLACK;
stack[ht - 1]->link[1] = rPtr->link[0];
rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) {
root = rPtr;
        } else {
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
dir[ht] = 0;
stack[ht] = stack[ht - 1];
stack[ht - 1] = rPtr;
ht++;

rPtr = stack[ht - 1]->link[1];
       }

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
rPtr->color = RED;
       } else {
if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
qPtr = rPtr->link[0];
rPtr->color = RED;
qPtr->color = BLACK;
rPtr->link[0] = qPtr->link[1];
qPtr->link[1] = rPtr;
rPtr = stack[ht - 1]->link[1] = qPtr;
        }
rPtr->color = stack[ht - 1]->color;
stack[ht - 1]->color = BLACK;
rPtr->link[1]->color = BLACK;
stack[ht - 1]->link[1] = rPtr->link[0];
rPtr->link[0] = stack[ht - 1];
if (stack[ht - 1] == root) {
root = rPtr;
        } else {
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
break;
       }
     } else {
rPtr = stack[ht - 1]->link[0];
if (!rPtr)
break;
```

Space for learners:

```
if (rPtr->color == RED) {
stack[ht - 1]->color = RED;
rPtr->color = BLACK;
stack[ht - 1]->link[0] = rPtr->link[1];
rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root) {
root = rPtr;
        } else {
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
dir[ht] = 1;
stack[ht] = stack[ht - 1];
stack[ht - 1] = rPtr;
ht++;

rPtr = stack[ht - 1]->link[0];
      }
if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
rPtr->color = RED;
      } else {
if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
qPtr = rPtr->link[1];
rPtr->color = RED;
qPtr->color = BLACK;
rPtr->link[1] = qPtr->link[0];
qPtr->link[0] = rPtr;
rPtr = stack[ht - 1]->link[0] = qPtr;
        }
rPtr->color = stack[ht - 1]->color;
stack[ht - 1]->color = BLACK;
rPtr->link[0]->color = BLACK;
stack[ht - 1]->link[0] = rPtr->link[1];
rPtr->link[1] = stack[ht - 1];
if (stack[ht - 1] == root) {
root = rPtr;
        } else {
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
break;
      }
    }
ht--;
  }
 }
```

Space for learners:

```
}


/* Inorder traversal of the tree */

voidinorderTraversal(RBtree *node) {
if (node) {
inorderTraversal(node->link[0]);
printf("%d  ", node->info);
inorderTraversal(node->link[1]);
  }
return;
}

/* Main Block */

int main() {
intch, val;
while (1) {
printf("1. Insertion\n");
printf("2. Deletion\n");
printf("3. Traverse\n");
printf("4. Exit\n");
printf("Enter your choice:");
scanf("%d", &ch);
switch (ch) {
case 1:
printf("Enter the element to insert:");
scanf("%d", &val);
insertion(val);
break;
case 2:
printf("Enter the element to delete:");
scanf("%d", &val);
deletion(val);
break;
case 3:
inorderTraversal(root);
printf("\n");
break;
case 4:
exit(0);
default:
printf("Not available\n");
break;
   }
printf("\n");
```

Space for learners:

```
  }
return 0;
}
```

Space for learners:

CHECK YOUR PROGRESS

1. Multiple Choice Questions

(i) Why do we need a binary search tree which is height balanced?

(a) To avoid formation of skew tree

(b) To save memory

(c) To attain faster memory access

(d) To simplify storing

(ii) What is the maximum height of an AVL tree with 7 nodes? Assume that the height of the tree with single node is 0.

(a) 2

(b) 3

(c) 4

(d) 5

(iii) Given an empty AVL tree, how would you construct AVL tree when a set of numbers are given without performing any rotation?

(a) Just build the tree with the given input.

(b) Find the median of the set of elements given, make it as root and construct the tree.

(c) Use trial and error method

(d) Use dynamic programming to build the tree.

(iv) What is the maximum difference in heights between the leafs of an AVL tree is possible?

(a) log(n) where n is the number of nodes

(b) n where n is the number of nodes

(c) 0 or 1

(d) At most 1

(v) Why to prefer Red Black trees over AVL trees?

(a) Because Red Black Tree is more rigidly balanced.

(b) AVL tree store balance factor in every node which costs space.

(c) AVL tree fails at scale

(d) Red Black tree is more efficient.

(vi) Why do we impose restrictions like

. root property is black

. every leaf is black

. children of red node are black

. all leaves have same black

(a) to get logarithm time complexity

(b) to get linear time complexity

(c) to get exponential time complexity

(d) to get constant time complexity

(vii) What are the operations that could be performed in O(log n) time complexity by Red Black tree.

(a) Insertion, deletion, finding predecessor, successor

(b) Only insertion

(c) Only finding predecessor, successor

(d) For sorting.

(viii) Which of the following is an application of Red Black Tree?

(a) Used to store strings efficiently

(b) Used to store integers efficiently

(c) Can be used in process scheduler, maps, sets

(d) For efficient sorting.

Space for learners:

(ix) Why Red Black trees are preferred over hash tables though hash tables have constant time complexity?

(a) No, they are not preferred.

(b) Because they can be implemented using trees

(c) Because they are balanced.

(d) Because of resizing issues of hash tables and better ordering in Red Black Trees

(x) How can you save memory when storing colour information in Red Black Trees?

(a) Using another array with colour of each nodes.

(b) Storing colour information in the node structure.

(c) Using least significant bit of one of the pointers in the node for colour information.

(d) Using negative and positive numbering.

2. Fill in the following blanks:

(i) In _____, balance factor of a node is the difference between height of left and right sub-tree.

(ii) There are _____ numbers of rotations are available at AVL tree.

(iii) The worst-case running time to search for an element in an AVL tree with $n*2^n$ element is_____.

(iv) An important quantitative measure of the complexity of a binary tree is its _____. It also provides a measure of the average depth of all the nodes in the tree.

(v) The worst-case time complexity of AVL tree is better in comparison to binary search tree for_____, _____, _____Operations.

(vi) In a Red Black Tree the restrictions are imposed to get _____ time complexity.

(vii) While inserting into _____tree, insertions are done at a leaf and will replace an external node with an internal node with two external children.

(viii) The Red Black Tree is a binary search tree whose leaves are _____ nodes.

(ix) The number of black nodes from the root to a node is the node's _____ _____; the uniform number of black

Space for learners:

nodes in all paths from root to the leaves is called the _____ _____of the Red Black Tree.

(x) A Red Black tree guarantees _____ time for searching.

Space for learners:

2.5 SUMMING UP

- An AVL tree is a self-balanced binary search tree where the difference between left and right sub-tree cannot be more than 1 for all the nodes.
- The AVL tree controls the height of the binary search tree and it prevents it from becoming skewed. When a binary search tree becomes skewed, it is the worst-case O(n) for all the operations. By using balance factor, AVL tree imposes a limit on the binary tree and keeps all the operations at O(log n).
- After performing insertion and deletion operation the binary search tree often become unbalance. To bring the binary search tree back to balance several rotation techniques need to apply. Basically, there are two types of rotation technique is applied and these are *Single Rotation* and *Double Rotation.* Again, there are two types of *Single Rotation- Left Rotation* and *Right Rotation* and two types of *Double Rotation – Left Right Rotation* and *Right Left Rotation.*
- *Left Rotation* is applied when node is inserted in the right sub-tree and *Right Rotation* is applied when node is inserted in the left sub-tree. *Left Right Rotation* is applied when the left sub-tree of a tree is right heavy and *Right Left Rotation* is applied when the right sub-tree of a tree is left heavy.
- AVL trees are most commonly used for in-memory sorts of sets and dictionaries.
- AVL trees are also used extensively in database applications in which insertion and deletions are fewer but there is frequent searching for data is required.
- AVL Tree is used in applications that required improve searching apart from the database applications.

- The Red Black Tree is a self-balancing binary search tree in which each node contains an extra bit for denoting colour of the node, either red or black.
- The root of the Red Black Tree is always Black. Two adjacent nodes cannot be Red. A Red node cannot have a Red parent or Red Child.
- The path from a node including the root node to any of its descendant's NULL nodes has the same number of Black nodes.
- A Red Black Tree is a particular implementation of a self-balancing binary search trees and nowadays it seems to be the most popular choice of implementation.
- The process scheduler in Linux uses Red Black Trees. The Completely Fair Scheduler (CFS) is the name of a process scheduler which was merged into the 2.6.23 release of the Linux kernel. It handles CPU resource allocation for executing process and aims to maximize overall CPU utilization while also maximizing interactive performance.

Space for learners:

2.6 ANSWERS TO CHECK YOUR PROGRESS

1.

(i) (a), (ii) (b), (iii) (b), (iv) (a), (v) (b), (vi) (a), (vii) (a), (viii) (c), (ix) (d), (x) (c)

2.

(i) AVL tree, (ii) four, (iii) O(log n), (iv) Average Path Length, (v) Search, Insert, Delete, (vi) logarithmic, (vii) Red Black, (viii) external, (ix) black depth black height, (x) O(log n).

2.7 POSSIBLE QUESTIONS

Short answer type questions:

1. What is an AVL tree?
2. Find the minimum number of nodes required to construct an AVL tree of height 3.

3. Mentions the rotations of AVL tree.
4. Write a short note on AVL tree application.
5. Discuss the different types of rotations available in AVL tree.
6. What are Red Black Trees? What problem they do solve?
7. How are Red Black Trees maintained in a balanced state?
8. Discuss the characteristics of Red Black Tree.
9. What are the rules to be followed while performing insertion and deletion operation in Red Black Tree?
10. How can you fix rule violations when a node is inserted or deleted from a Red Black Tree?

Long answer type questions:

1. What is an AVL Tree? How do you define the height of it? Explain about balance factor associated with a node of an AVL Tree.
2. Build an AVL tree for the following data. Show the step-by-step construction 25,12,17,30, 15, 14, 37, 27, 40, 29, 28.
3. What is the maximum height of a Red-Black Tree with 14 nodes? (Hint: The black depth of each external node in this tree is 2.) Draw an example of a tree with 14 nodes that achieves this maximum height.
4. Why can't a Red-Black tree have a black node with exactly one black child and no red child?

Space for learners:

2.8 REFERENCES AND SUGGESTED READINGS

1. Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest′ "*Introduction to Algorithms*" 3rd Edition, The MIT Press, Cambridge, Massachusetts London, England.

UNIT 3: MULTI-WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES

Space for learners:

Unit Structure:

3.1 INTRODUCTION

In this unit you will learn about some another advanced binary search tree such as Multi way Search Tree, 2-3 trees and Splay Trees. A multi way search tree is a tree that can have more than two children. This tree is generalized version of binary search tree where each node contains multiple elements. This tree is also called m-way tree. In an m-way tree of order m, each node contains a maximum of m-1 elements and m children. A 2-3 tree is a tree data structure in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children. Splay tree is also another type of self-balancing binary search tree. The main idea of splay tree is to bring the recently accessed item to the root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use the locality of reference.

Space for learners:

3.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- *understand* the fundamental concepts of multi way tree, 2-3 tree and Splay tree.
- *know* how B-Tree and B+ tree are specialized multi way tree
- *know*how to perform insertion, deletion and searching in multi way tree.
- *know* how insertion and deletion can be performed in 2-3 tree.
- *understand* the properties of the Splay Tree.
- *learn* about Rotation in Splay Tree.

3.3 MULTI-WAY SEARCH TREE

3.3.1 Introduction

A multi way (M-way) search tree is similar to binary tree. The only difference is that a binary tree can have 0, 1 or 2 number of child nodes whereas in an M-way search tree it has M-1 values per nodes and M-sub-trees. In this type of tree, M is called degree of the tree.

An M-way tree is defined as a tree that can have two or more children. If an M-way tree can have maximum *m* children, then the tree is called multi way tree of order m (or an m-way tree). As with other trees that have been studied, the nodes in an m-way tree will be made up of m-1 key fields and pointers to the children. By definition an m-way tree in which following condition should be satisfied-

- Each node is associated with m children and m-1 key field.
- The keys in each node are arranged in ascending order.
- The keys in the first j children are less than the j^{th} key.
- The keys in the last m-j children are higher than the j^{th} key.

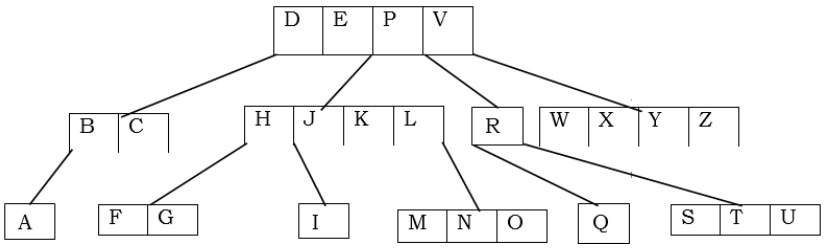


Figure 3.1: A multi way tree of order 5

The ordering principle in multi way search tree holds the principle of binary search tree such as anything to the left is smaller than its parent key, to the right is larger than its parent key. If more than one key in the node, the pointer between two keys will point to values between the two keys. If more than one key in a node, the keys will be is sequential order. Multi way search tree provides fast information retrieval and update. However, they also have same problems that binary search trees had- they become unbalanced, which means that the construction of the tree becomes of vital importance.

A B-tree is an extension of an M-way search tree. Besides having all the properties of an M-way search tree, it has some properties of its own, these mainly are:

- All the leaf nodes in a B tree are at the same level.
- All internal nodes must have M/2 children.
- If the root node is a non-leaf node, then it must have at least two children.

Space for learners:

- All node except the root node, must have at least [M/2]-1 keys and at most M-1 keys.

Space for learners:

3.3.2 Searching in a Multi-Way Search Tree

If we want to search for a value in say N in a M-way search tree and currently we are at a node that contains key values from M1, M2, M3……..Mk. Then in total 4 cases are possible to deal with this scenario, these are:

- If N<M1, then we need to recursively traverse the left sub-tree of M1.
- If N>Mk, then we need to recursively traverse the right sub-tree of Mk.
- If N=Mi, for some I, then we are done, and can return.
- Last and only remaining case is that when for some I we have Mi<N<M(i+1) then in this case we need to recursively traverse the sub-tree that is present in between Mi and M(i+1).

A pictorial representation of an M-way search tree is shown below-

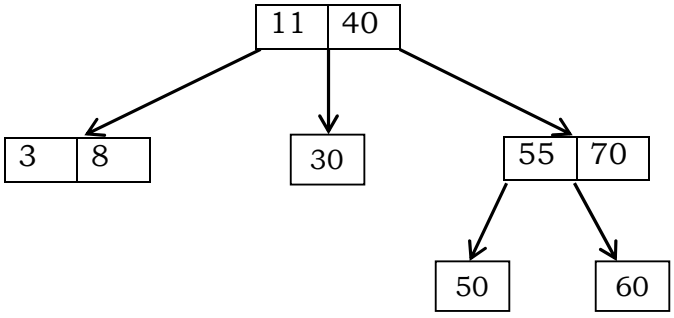


Figure 3.2: An M-way search tree (it is a 3-way search tree)

From the Figure 3.2, say we want to search for a node having key(N) equal to 60. Then, considering the above cases, for the root node, the second condition applies, and (60>40) and hence we move on level down to the right sub-tree of 40. Now, the last condition is valid only, hence we traverse the sub-tree which is in between the 55 and 70. And finally, while traversing down, we have our value that we are looking for.

The structure of a node in an M-way search tree is given below

typedefstruct node

```
{
    intnum;
    intval[MAX+1];
    struct node *child[MAX+1];
}mTree;
```

- Here ***num*** represents the number of children that a node has.
- The values of a node stored in the array ***val***.
- The addresses of child nodes are stored in the ***child*** array.
- The ***MAX*** macro signifies the maximum number of values that a particular node can contain.

The coding for searching in M-way search tree is given below

```
mTree *search(intval, mTree *root, int *pos)
{
        if(root==NULL)
                return NULL;
        else{
                if(searchnode(val, root,pos))
                        return root;
                else
                        return search(val, root->child[*pos],pos);
        }
}

intsearchnode(intval, mTree *n, int *pos)
{
        if(val<n->val[1])
        {
                *pos=0;
                Return 0;
        }
        else
        {
                *pos=n->num;
                while((val<n->val[*pos])&& *pos>1)
                        (*pos)--;
                if (val==n->val[*pos])
```

Space for learners:

```
                return 1;
            else
                return 0;
        }
    }
```

3.3.3 Insertion into a Multi-Way Search Tree

The insertion in an M-way search tree is similar to binary trees but there should not be more than m-1 elements in a node. If the node is full then a child node will be created to insert in the further elements. An example is given below to demonstrate the insertion into an m-way search tree.

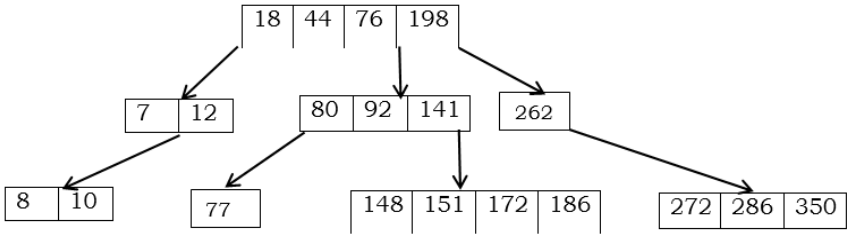


Figure 3.3: An M-way Search Tree (5-way search tree)

To insert a new value into an m-way search tree we precede the same way as one would in order to search for the element. To insert 6 into the m-way search tree as shown in Figure 3.3, we proceed to search for 6 and find that we fall off the tree at the node [7, 12] with the first child node showing a null pointer. Since the node has only 2 keys and a 5-way search tree can accommodate up to 4 keys in a node, 6 is inserted into the node like [6,7,12].

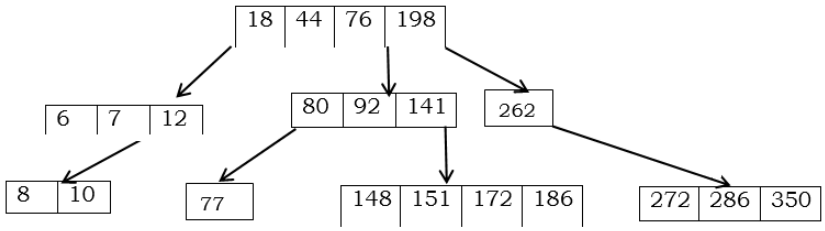


Figure 3.4: After inserting 6 into the tree

Space for learners:

But to insert 146, the node [148, 151, 172, 186] is already full, hence we have to open a new child node and insert 146 into it as per the Figure 3.4 shown below-

Space for learners:

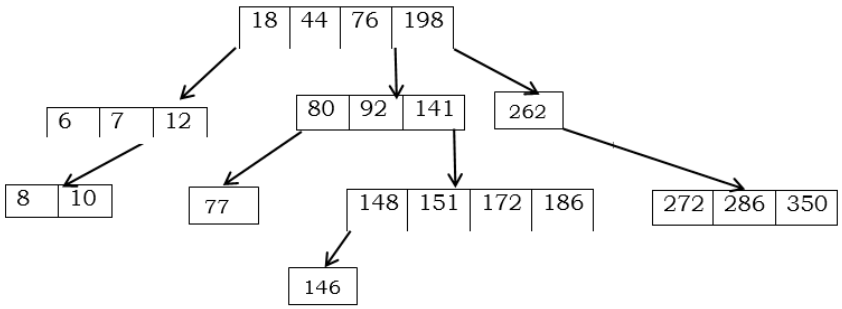


Figure 3.5: After inserting 146 into the tree

The coding for insertion in an M-way search tree is given below

```
mTree *insert(intval, mTree *root)
{
        int i;
        mTree *c, *n;
        int flag;
        flag=setval(val, root, &I, &c);
        if(flag)
        {
                n=(mTree *)malloc(sizeof(mTree));
                n->num=1;
                n->val[1]=i;
                n->child[0]=root;
                n->child[1]=c;
                return n'
        }
        return root;
}
intsetval(intval, mTree *n, int *p, mTree **c)
          {
                int k;
```

```
            if(n==NULL)
            {
                  *p=val;
                  *c=NULL;
                  return 1;
            }
            else
            {
                  if (searchnode(val, n, &k))
                        printf("\nKey value already exists\n");
                  if(setval(val, n->child[k],p,c)
                  {
                        fillnode(*p,*c,n,k);
                        return 0;
                  }
                  else
                  {
                        split(*p,*c, n,k,p,c);
                        return 1;
                  }
            }
            return 0;
      }
}

voidfillnode(intval, mTree *c, Mtree *n, int k)
{
      int i;
      for(i=n->num; i>k;i--)
      {
            n->val[i+1]=n->val[i];
```

Space for learners:

```
                n->child[i+1]=n->child[i];
        }
        n->val[k+1]=val;
        n->child[k+1]=c;
        n->num++;
 }

void split (intval, mTree *c, mTree *n, intk,int *y, mTree **newnode)
        {
                int I, mid;
                if(k<=MIN)
                        mid=MIN;
                else
                        mid=MIN+1;
                *newnode=(mTree *)malloc(sizeof(Mtree));
                for(i=mid+1;i<=MAX;i++)
                {
                        (*newnode)->val[i-mid]=n->val[i];
                        (*newnode)->child[i-mid]=n->child[i];
                }
                (*newnode)->num=MAX-mid;
                n->num=mid;
                if(k<=MIN)
                        fillnode(val,c,n,k);
                else
                        fillnode(val, c, *newnode, k-mid);
                *y=n->val[n->num];
                (*newnode)->child[0]=n->child[n->num];
                n->num--;
        }
```

Space for learners:

3.3.4. Deletion in Multi-Way Search Tree

Space for learners:

Deletion operation in an M-way search tree must observe the rule that the all the leaves are at the same level after deleting an element. Each node must have between 1 and m-1 keys and it must remain a search tree after deletion. If a deletion removes all the keys from a node, sibling nodes must be merged that means a key must be removed from a parent. There are several ways of doing deletions. In some implementations, adopting from a sibling is allowed. A deletion may even force a height reduction but it is avoided if possible, since an insertion may again force a height increase.

Let K be a key to be deleted from the m-way search tree and A_i and A_j pointers to the sub-tree. To delete the key we proceed as one would to search for the key. There are several cases for deletion.

- If (A_i=A_j=NULL) then delete K
- If (A_i! =NULL, A_j =NULL) then choose the largest of the elements K' in the child pointed to by A_i, delete the key K' and replace K by K'.
- The deletion of K' may call for subsequent replacements and therefore deletion in a similar manner, to enable the key K' move up the tree.
- (A_i=NULL, A_j!=NULL) then choose the smallest of the key element K" from the sub-tree pointed to by A_j, delete K" and replace K by K".
- Again, deletion of K" may trigger subsequent replacements and deletions to enable K" move up the tree.
- If (A_i!=NULL A_j!=NULL) then choose either the largest of the key elements K' in the sub-tree pointed to by A_i to replace K.
- As mentioned above, to move K' or K" up the tree it may call for subsequent replacements and deletions.

An example is given below to illustrate the above mentioned points.

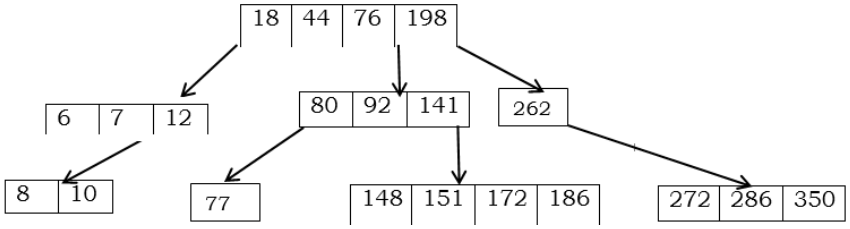


Figure 3.6: An m-way search tree

- To delete 151, we search for 151 and observe that in the leaf node [148, 151, 172, 186] where it is present, both its left sub-tree pointer and right sub-tree pointer are such that (A_i=A_j=NULL).
- We therefore simply delete 151 and the node becomes [148, 172, 186]. Deletion of 92 also follows similar process.
- To delete 262, we find its left and right sub-tree pointers A_i and A_j respectively, are such that A_i=NULL and A_j!=NULL.
- Hence we choose the smallest element 272 from the child node [272, 286, 350], delete 272 and replace 262 with 272. To delete 272 the entire deletion procedure needs to be observed again.
- To delete 12, we find the node [7, 12] accommodates 12 and the key satisfied (A_i!=NULL, A_j=NULL).
- Hence we choose the largest of the keys from the node pointed by A_i viz. 10 and replace 12 with 10.

The coding for deletion from an M-way search tree is given below

```
mTree *del(intval, mTree *root)
{
        mTree *temp;
        if(!delhelp(val, root){
                printf("\nThe value %d not found.\n",val);
        }
        else {
                if(root->num==0) {
                        temp=root;
                        root=root->child[0];
                        free(temp);
```

Space for learners:

```
            }
        }
        return root;
            }
            intdelhelp(intval, mTree *root)
            {
                int i;
                int flag;
                if(root==NULL)
                    return 0;
                else {
                    flag=searchnode(val, root, &i);
                    if(flag) {
                        if(root->child[i-1]) {
                            successor(root, i);
                            flag=delhelp(root-
>val[i], root->child[i])
                            if(!flag){
        printf("\nThe value %d not found.\n",val);
                    }
                }
                else {
                    clear(root, i);
            }
            else {
                flag=delhelp(val, root->child[i]);
            }
            if( root->child[i]!=NULL){
                if(root->child[i]]->num< MIN)
                    restore(root, i);
            }
```

Space for learners:

```
        return flag;
}
        }
        void clear(mTree *m, int k)
        {
                int i;
                for(i=k+1; i<=m->num;i++)
                {
                        m->val[i-1]=m->val[i];
                        m->child[i-1]=m->child[i];
                }
                m->num--;
        }
        void successor(mTree *m, int i)
        {
                mTree *temp;
                temp=m->child[i];
                while(temp->child[0])
                        temp=temp->child[0];
                m->val[i]=temp->val[i];
        }
        void restore( mTree *m, int i)
        {
                if(i==0) {
                        if(m->child[1]->num> MIN)
                                leftshift(m,1);
                        else
                                merge(m,1);
                }
                else {
                        if(i==m->num) {
```

Space for learners:

```
                        if(m->child[i-1]->num>MIN)
                                rightshift(m,i);
                        else
                                merge(m, i);
                }
                else {
                if(m->child[i - 1]->num> MIN)
        rightshift(m, i);
        else{
        if(m->child[i + 1]->num> MIN)
        leftshift(m, i + 1);
        else
        merge(m, i);
        }
        }
        }
}

        voidrightshift( mTree *m, int k)
        {
                int i;
                mTree *temp;
                temp = m->child[k];
                for(i=temp->num;i>0;i--)
                {
                        temp->val[i+1]=temp->val[i];
                        temp->child[i+1]=temp->val[i+1];
                }
                temp->child[1]=temp->child[0];
                temp->num++;
                temp->val[1]=m->val[k];
                temp=m->child[k-1];
                m->val[k]=temp->val[temp->num];
                m->child[k]->child[0]=temp->child[temp->num];
                temp->num--;
```

Space for learners:

```
}
voidleftshift(mTree *m, int k)
{
        int i;
        mTree *temp;
        temp=m->child[k-1];
        temp->num++;
        temp->val[temp->num]=m->val[k];
        temp->child[temp->num]=m->child[k]->child[0];
        temp=m ->child[k];
        m->val[k]=temp->val[1];
        temp->child[0]=temp->child[1];
        temp->num--;
        for(i=1;i<=temp->num;i++) {
                temp->val[i]=temp->val[i+1];
                temp->child[i]=temp->child[i+1];
        }
}

void merge(mTree *m, int k)
{
        int i;
        mTree*temp1, *temp2;
        temp1=m->child[k];
        temp2=m->child[k-1];
        temp2->num++;
        temp2->val[temp2->num]=m->val[k];
        temp2->val[temp2->num]=m->child[0];
        for(i=0;i<=temp->num;i++) {
                temp2->num++;
                temp2->val[temp2->num]=temp1->val[i];
```

Space for learners:

```
                                        temp2->child[temp2->num] = temp1-
>child[i];
        }
        for(i = k; i < m->num; i++) {
        m->val[i] = m->val[i + 1];
        m->child[i] = m->child[i + 1];
        }
        m->num--;
        free(temp1);
        }
```

3.4 2-3 TREES

3.4.1. Introduction

A 2-3 Tree is a tree data structure where every node with children has either two children one data element or three children and two data elements. A node with two children is called a 2-Node and a node with three children is called a 3- Node. A 4-Node, with three data elements, may be temporarily created during manipulation of the tree but is never persistently stored in the tree. A 2-3 tree is a B-Tree of order 3. Nodes on the outside of the tree have no children and one or two data elements. The 2-3 Tree was invented by John Hopcroft in the year 1970. 2-3 Trees are required to be balanced so that each leaf must be at the same level. It follows that each right, centre and left sub-tree of a node contains the same or close to the same amount of data.

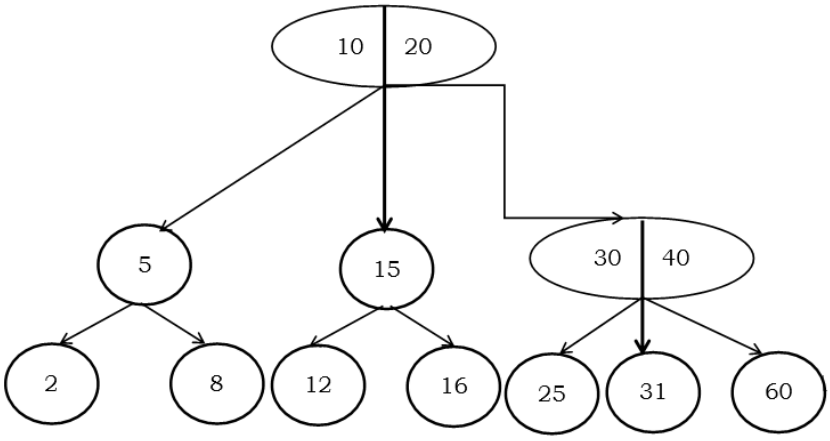


Figure 3.7:A 2 -3 Tree

Space for learners:

A 2-3 Tree must resemble the following properties-

- Every internal node in the tree is a 2-node or a 3 – node, means it has either one value or two values.
- A node with one value is either a leaf node or has exactly two children. *Values in left sub tree < value in node < values in right sub tree.*
- A node with two values is either a leaf node or has exactly 3 children. It cannot have 2 children. *Values in left sub tree < first value in node < values in middle sub tree < second value in node < value in right sub tree.*
- All leaf nodes are at the same level.

Space for learners:

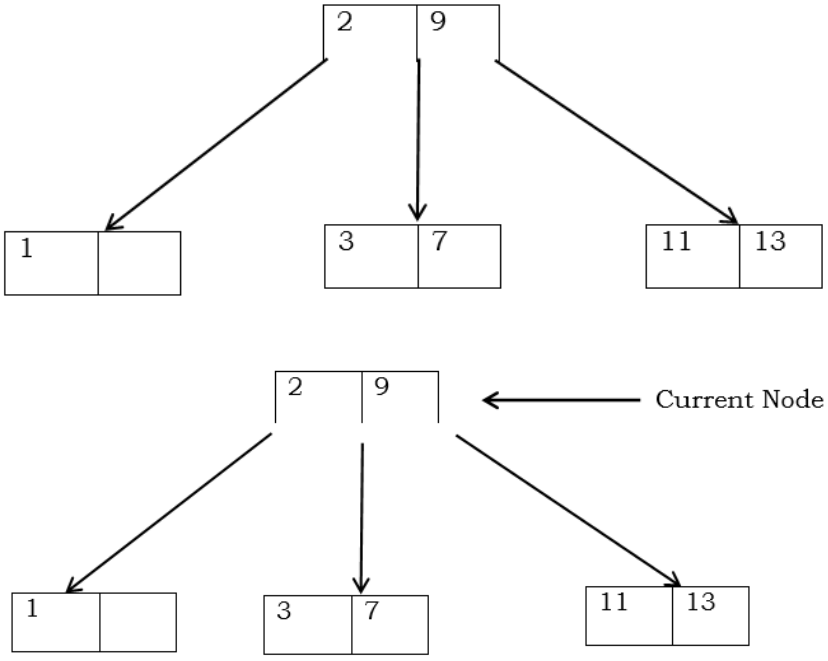
3.4.2 Searching Operation in A 2-3 Tree

Searching for an item in a 2-3 Tree is same as searching an element in a binary search tree. Since the data elements in each node are treated as ordered, a search function will be directed to be correct sub-tree and eventually to the correct node which consists of the item.

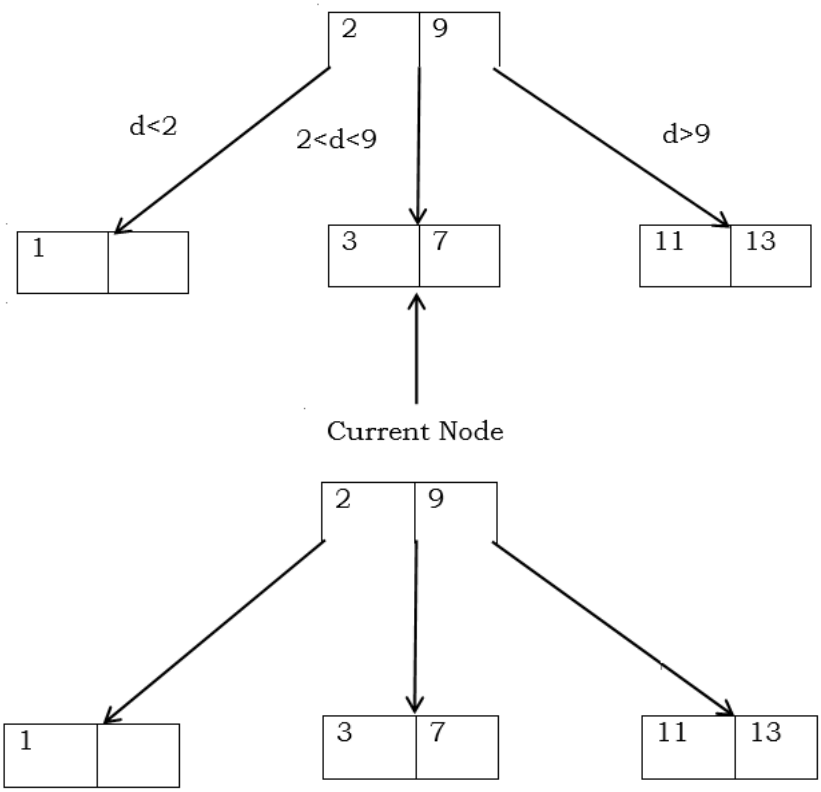
- Let *T* be a 2-3 Tree and *d* be the element we want to search. If *T* is empty, then *d* is not in *T* and we will abort the searching.
- Let *r* be the root of the *T*.
- Let *r* is a leaf. If *d* is not in *r*, as a result *d* is not in *T*. Otherwise, *d* is in*T*. In particular, *d* can be found at a leaf node. We need no further steps and can conclude the searching operation.
- Suppose *r* is treated as a 2-node with left child *L* and right child *R*. Let *e* be treated as the data element in *r.* in this situations there are three cases:
- If *d* is equal to *e*, then we've found *d* in *T* and can conclude the search operation.
- If *d* is less than *e*, then set *T* to *L,* which is by definition is a 2-3 Tree, and return back to Step 2.
- If *d* is greater than *e*, then set *T* to *R,* and return back to Step 2.

- Let *r* is a 3-node with left child *L*, middle child *M*, and right child *R*. Let *a* and*b* be treated as the two data elements of *r*, where a<b. There are four cases-
    - If *d* is equal to *a* or*b*, then *d* is in *T* and we are performed.
    - If *d* is less than *a*, then set *T* to *L* and return back to step 2.
    - If *a*is less than *d* and *d* is less than *b*, then set *T* to *M* and return back to step 2.
    - If *d* is greater than *b*, then set *T* to *R* and return go back to step 2.

Let us look into the following example. We are searching 5 in the following 2-3 trees.



Space for learners:



5 not found. Return False

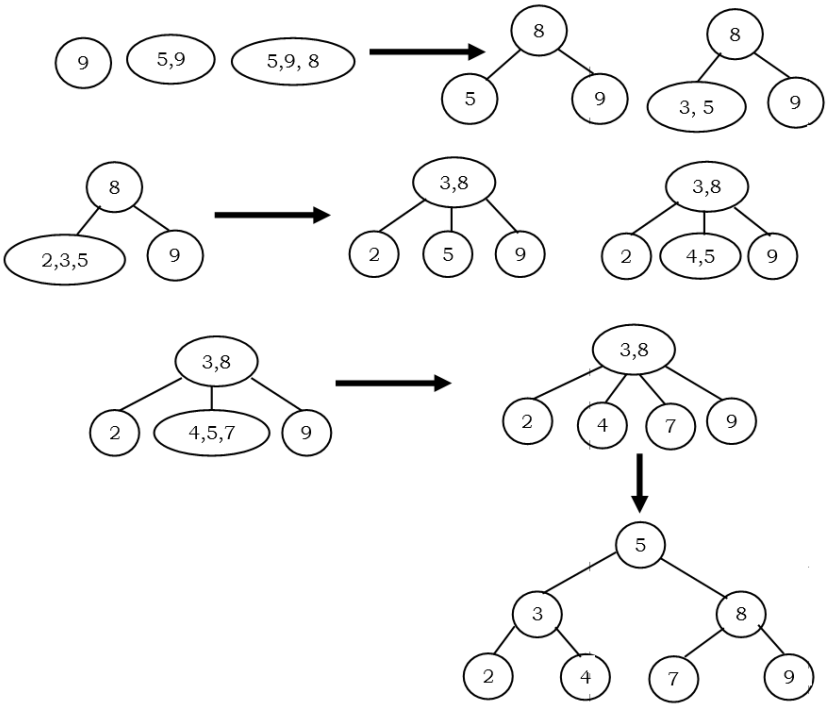
3.4.3 Insertion Operation in A 2-3 Tree

The insert operation takes care of the balanced property of a 2-3 tree. The insertion operation algorithm into a 2-3 tree is quite different from the insertion operation in a binary search tree. In a 2-3 tree, the algorithm will be as follows:

1. If the tree is empty, create a node and put value into the node.
2. Otherwise find the leaf node, where the value belongs.
3. If the leaf node as only one value, put the new value into the node.
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent has then three values, continue to split and promote, forming a new root node if necessary.

Space for learners:

The following example should help you to better understanding of insertion algorithm. Let us start insert 9,5,8,3,2,4,7 starting from an empty tree.



3.4.4 Deletion Operation in a 2-3 Tree

Deleting an element from a 2-3 tree is similar to insertion. There is a special case when the tree T is just a single node containing data element d. In this case, the tree is made empty. In other cases, the parent of the node to be deleted is found, then the tree is fixed up, if required so that it still be a 2-3 tree. Once the parent of the node n to be deleted is just found, there are two cases depending on how many children n has –

- **If n has 3 children**

  Remove the child with value d, then fix the left value, middle value and n's ancestors' left value and middle value if necessary.

- **If n has 2 children**
  - If n is the root of the tree, then remove the node containing d. Replace the root node with other children.

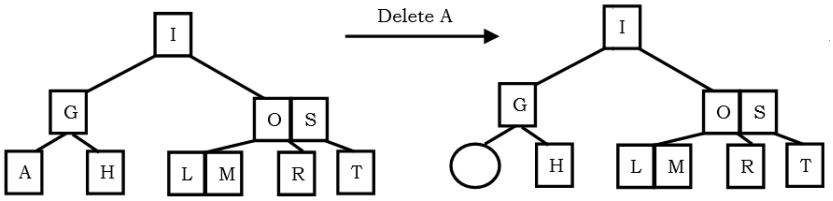
Space for learners:

- If n has a left or right sibling with 3 kids, then
  1. Remove the node containing d.
  2. Use one of the sibling's children.
  3. Fix left value, middle value of n and n's sibling and ancestors as needed.
- If n's siblings have only 2 children, then:
  1. remove the node containing **d**
  2. make n's remaining child a child of n's sibling
  3. fix left value and middle value fields of n's sibling as needed
  4. Remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed.

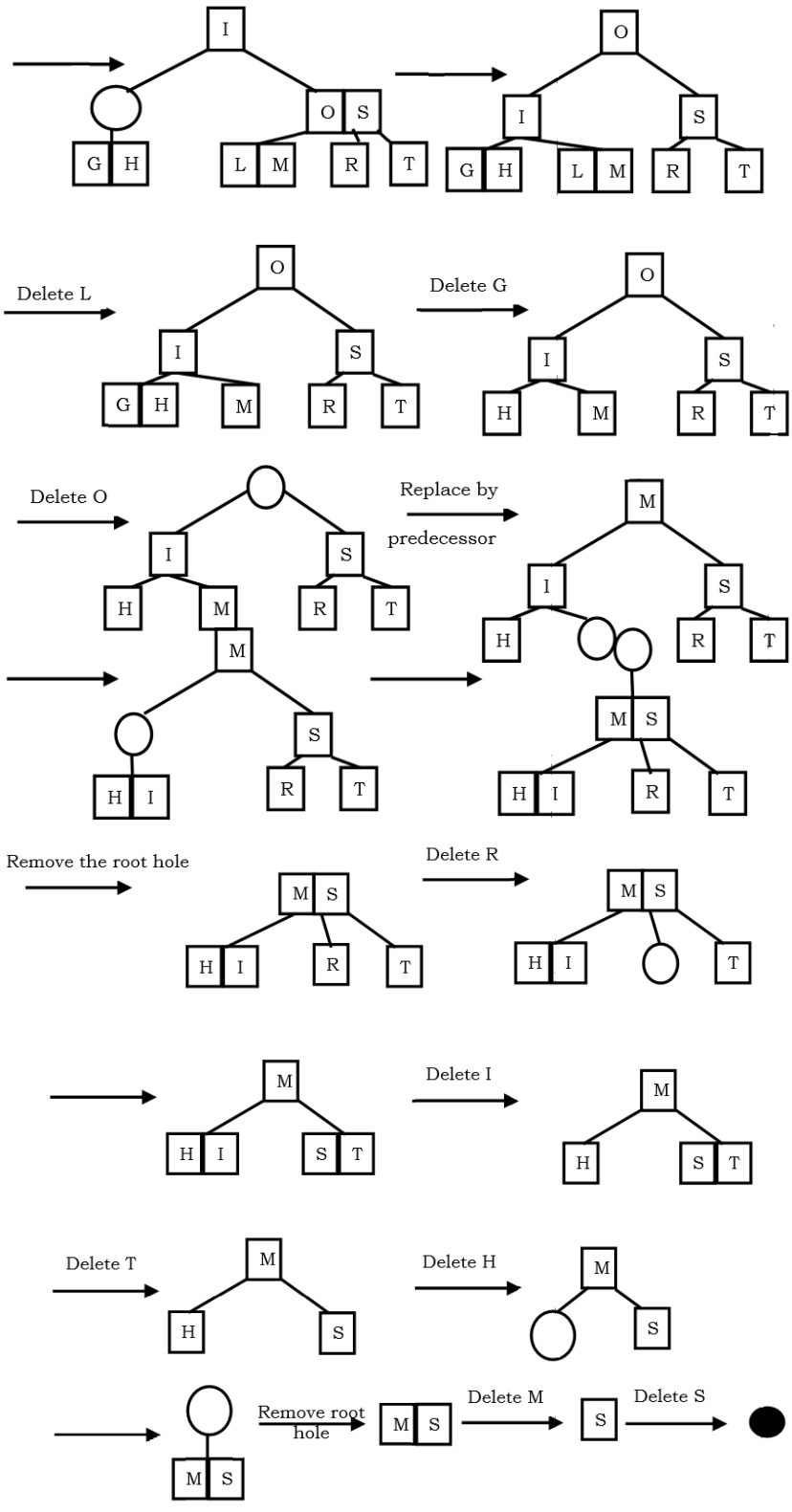
The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing left Max and middle Max fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).

So the total time is 2 * height-of-tree = O(log N).

An example of deletion is shown below where letter by letter deletion of the letters **A L G O R I T H M S** from the tree that is formed after inserting them.



Space for learners:



Space for learners:

3.5 SPLAY TREE

3.5.1 Introduction

A **splay tree** is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality. A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. Splay trees can provide good performance in this situation.

Importantly, splay trees offer amortized $O(\lg n)$ performance; a sequence of M operations on an n-node splay tree takes $O(M \lg n)$ time.

A splay tree is a binary search tree. It has one interesting difference, however: whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant. If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.

We have already seen a way to move an element upward in a binary search tree: tree rotation. When an element is accessed in a splay tree, tree rotations are used to move it to the top of the tree. This simple algorithm can result in extremely good performance in practice. Notice that the algorithm requires that we be able to update the tree in place, but the abstract view of the set of elements represented by the tree does not change and the rep invariant is maintained. This is an example of a benign side effect, because it does not change the value represented by the data structure.

Space for learners:

3.5.2 Rotations in Splay Tree

Space for learners:

There are three kinds of tree rotations that are used to move elements upward in the tree. These rotations have two important effects: they move the node being splayed upward in the tree, and they also shorten the path to any nodes along the path to the splayed node. This latter effect means that splaying operations tend to make the tree more balanced.

When a node *x* is accessed, a splay operation is performed on *x* to move it to the root. To perform a splay operation, we carry out a sequence of *splay steps*, each of which moves *x* closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Each particular step depends on three factors:

- Whether *x* is the left or right child of its parent node, *p*,
- Whether *p* is the root or not, and if not
- Whether *p* is the left or right child of *its* parent, *g* (the *grandparent* of x).

It is important to remember to set *gg* (the *great-grandparent* of x) to now point to x after any splay operation. If *gg* is null, then x obviously is now the root and must be updated as such.

There are three types of splay steps, each of which has two symmetric variants: left- and right-handed. For the sake of brevity, only one of these two is shown for each type. (In the following diagrams, circles indicate nodes of interest and triangles indicate sub-trees of arbitrary size.) The three types of splay steps are:

- Zig Rotation
- Zig-Zig Rotation
- Zig-Zag Rotation

Zig Rotation:

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example:

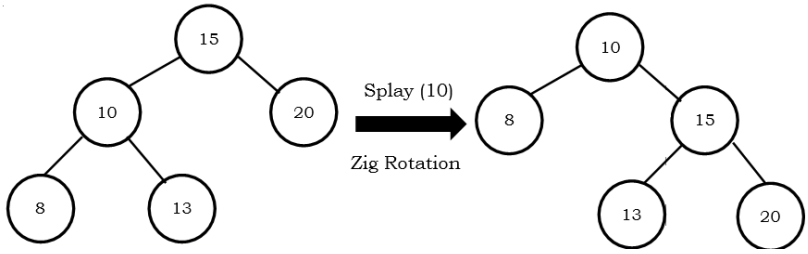


Figure 3.8Zig Rotation

Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example:

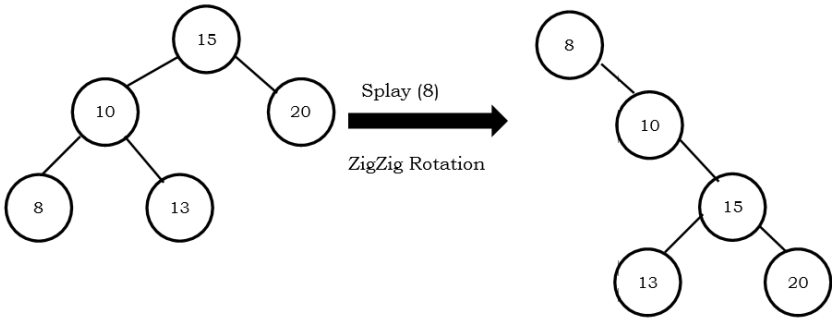


Figure 3.9: Zig-Zig Rotation

Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example:

Space for learners:

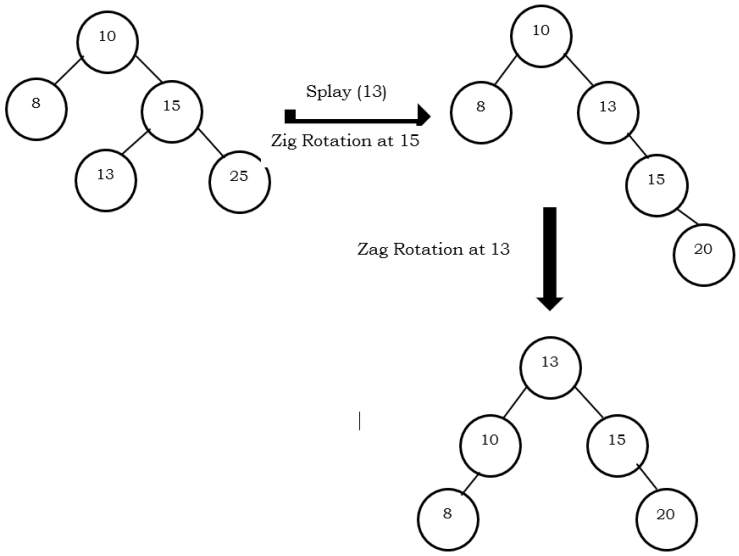


Figure 3.10 Zig-Zag Rotation

3.5.3 Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps:

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.

Step 4 - After insertion, **Splay** the **newNode**

3.5.4 Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

Space for learners:

Space for learners:

CHECK YOUR PROGRESS

1. Multiple Choice Questions

(i) A 2-3 tree is a specific form of_____
   (a) B – tree
   (b) B+ – tree
   (c) AVL tree
   (d) Heap

(ii) The height of 2-3 tree with n elements is _____
   a) between (n/2) and (n/3)
   b) (n/6)
   c) between (n) and log2(n + 1)
   d) between log3(n + 1) and log2(n + 1)

(iii) Which of the following the BST is isometric with the 2-3 tree?
   a) Splay tree
   b) AA tree
   c) Heap
   d) Red – Black tree

(iv) Which of the following is not true about the 2-3 tree?
   a) all leaves are at the same level
   b) it is perfectly balanced
   c) post-order traversal yields elements in sorted order
   d) it is B-tree of order 3

(v) What are Splay Trees?
   (a) Self-adjusting binary search tree
   (b) Self-adjusting binary tree
   (c) A tree with strings
   (d) A tree with probability distribution.

(vi) Which of the following property of Splay Tree is correct?
   (a) It holds probability usage of the respective sub-trees.
   (b) Any sequence of j operations starting from an empty tree with h nodes almost, takes O(jlogh) time complexity.
   (c) Sequence of operations with h nodes can take O(log h) time complexity.
   (d) Splay trees are unstable trees.

(vii) Why to prefer Splay Tree?
   (a) Easier to program.
   (b) Space efficiency
   (c) Easier to program and faster access to recently accessed items.
   (d) Quick searching.

(viii) Which of the following is an application of Splay Tree?
   (a) Cache Implementation
   (b) Networks
   (c) Send values
   (d) Receive values

(ix) What is the disadvantage of using Splay Tree?
   (a) Height of a splay tree can be linear when accessing elements in non-decreasing order.
   (b) Splay operations are difficult.
   (c) No significant disadvantage.
   (d) Splay tree performs unnecessary splay when a node is only being read.

(x) When we have red-black trees and AVL trees that can perform most of operations in logarithmic times, then what is the need for splay trees?
   (a) No. there is no special usage.
   (b) In real time it is estimated that 80% access is only to 20% data, hence most used ones must be easily available
   (c) Red black and AVL are not upto mark
   (d) They are just another type of self-balancing binary search trees

2. Fill up the blanks:

(i) Multi way search trees are not binary search tree because_____.

(ii) A Multi way search tree has n items. The number of external node for this tree is_____.

(iii) 2-3 tree is a specific form of _____.

(iv) For efficient searching of elements we prefer_____ data structure.

(v) An internal node in a 2-3 tree is said to be a _____ node if it has two data elements and _____children.

Space for learners:

(vi) The maximum number of children a 2-3 tree can have is_____.

(vii) Moving a node to the root is called _____ operation.

(viii) Self-adjusting binary search tree is called _____.

(ix) Splay trees have _____ complexity.

(x) _____ of a Splay tree can be linear when accessing elements in non-descending order.

Space for learners:

3.6 SUMMING UP

- The **m-way** search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements. In an m-Way tree of order **m**, each node contains a maximum of **m – 1**elements and m children. The goal of m-Way search tree of height h calls for O(h) no. of accesses for an insert/delete/retrieval operation. Hence, it ensures that the height **h** is close to **log_m(n + 1)**. The number of elements in an m-Way search tree of height **h** ranges from a minimum of **h** to a maximum of m^h*-1*. An m-Way search tree of n elements ranges from a minimum height of **log_m(n+1)** to a maximum of **n.**
- One of the advantages of using these multi-way trees is that **they often require fewer internal nodes than binary search trees to store items**. But, just as with binary search trees, multi-way trees require additional methods to make them efficient for all dictionary methods.
- Searching is much like a binary tree search, except you may have more than one key in a node. If the item you are searching for is less than the leftmost key, go left. If it is greater than the leftmost key, consider the next key; if it is between the two keys, follow the pointer to the right of the leftmost key. If it is greater than the second key, consider the next key, following the pointer to the left of the first key it is less than.
- A **2-3 Tree** is a multiway search tree. It's a *self-balancing* tree; it's always perfectly balanced with every leaf node at equal distance from the root node.Other than the leaf nodes, every node can be one of two types:

- **2-Node**: A node with a single data element that has two child nodes
- **3-Node**: A node with two data elements that has three child nodes

- Even though searching a 2-3 tree is not more efficient than searching a binary search tree, by following the node of a 2-3 tree to have three children, a 2-3 tree might be shorter than the shortest possible binary search tree.
- Maintaining the balance of a 2-3 tree is relatively simple than maintaining the balance of a binary search tree.
- Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree. In a splay tree, every operation is performed at the root of the tree.
- A splay tree is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality. A good example is a network router.

Space for learners:

3.7 ANSWERS TO CHECK YOUR PROGRESS

1.
(i) (a), (ii) (d), (iii) (b), (iv) (c), (v) (a), (vi) (b), (vii) (c) , (viii) (b), (ix) (a), (x) (b).

2.
(i) it can have more than two children, (ii) n+1, (iii) B-Tree, (iv) 2-3 tree, (v) Three Three, (vi) Six, (vii) Splay, (viii) Splay Tree, (ix) O(log n), (x) Height.

3.8 POSSIBLE QUESTIONS

Short answer type questions:

1. What are multi way trees?
2. Describe the operations can be performed on a multi-way search tree.
3. Explain the properties of 2-3 trees.

4. What are the probable cases of deletion operation in a 2-3 tree.
5. The keys of value N, N-1, N-2,.., 4, 3, 2, 1 are inserted in this order in a splay tree. What is the final configuration of the tree? What is the cost in Big-Oh notation of each insert operation?
6. What are the rotations available in Splay tree?

Long answer type questions:

1. Write down the algorithm for performing search operation in Multi-way search tree.
2. Write down the algorithm for performing insertion operation in Multi-way search tree.
3. Write down the algorithm for performing deletion operation in Multi-way search tree.
4. Discuss how insertion operation performed in 2-3 tree.
5. Discuss how deletion operation performed in 2-3 trees with example.
6. Create a 2-3 tree from the following list of data items 5, 6, 8, 21, 12, 30, 34, 27, 23, 4, 33, 7, 24, 9, 10, 11, 13, 38.
7. Explain with example, what are the different cases followed while inserting a node in a 2-3 Tree.
8. Explain with diagrams the rotations available in Splay tree.

Space for learners:

3.9 REFERENCES AND SUGGESTED READINGS

1. Dinesh P Mehta, Sartaj Sahni, "*Handbook of Data Structures and Applications"* 2nd Edition, O' Reilly Publication.
2. Peter Brass *"Advanced Data Structures Hardcover- Illustrated, 8, September 2008",* Cambridge Publisher.
3. Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest′ "*Introduction to Algorithms*" 3rd Edition, The MIT Press, Cambridge, Massachusetts London, England.

UNIT 4: HASHING

Unit Structure:

4.1 INTRODUCTION

In Data structure algorithms, generally we study how to retrieve an element from any data structure where it's stored. In Sequential search and binary search and all the search trees, searching time depends on number of elements and number of key comparisons. Linear search running time is proportional to O (n), while Binary search running time is proportional to O(log n). In a balanced binary search tree, running time can be guaranteed to be in O (log n). Here we will discuss a good searching approach where less key comparisons are involved and searching can be performed in constant time i.e. O(1). In this searching approach searching time is independent of the number of elements. Hashing is such a kind of approach. Before going to hashing we will discuss a data structure named direct access table. By considering an example we will describe the direct access table. Suppose we would like to store students' records keys using phone numbers.

Space for learners:

In the direct access table where we create an array and phone numbers will be considered as index in the array. If no phone number is present, an entry in the array is NIL. Otherwise the array entry stores pointers to records corresponding to phone numbers. In this approach, we can insert, delete and search in O(1) time. Time complexity wise this solution is the best among all. For example, to insert a phone number, we create a record with details of the given phone number, use the phone number as index and store the pointer to the created record in the table. The direct access table has many practical limitations. First of all it requires a huge extra space. Second problem is that an integer in a programming language may not store n digits.

We can overcome this problem in Hashing. In hashing we get O(1) search time on an average and O(n) in the worst case. Hashing is an improvement over Direct Access Table.

Space for learners:

4.2 UNIT OBJECTIVES

In this Chapter we will study the concepts of the following:

- understand the basic concept of hashing
- importance of hashing
- hash table and hash function
- different types of hash function
- collision resolution
- different types of collision resolution techniques

4.3 HASHING

Hashing is a technique to convert a range of key values into a range of indexes of an array. In other words we can say that hashing is a technique or process of mapping keys, values into the hash table by using a hash function. Hashing is also called the message digest function. It is widely used in the encryption and decryption of digital signatures.

Let us describe the concept by using an example.

Suppose we have to store 100 numbers of student information. Here each student has a unique roll number in the range 0-99 with name. Roll numbers will be considered as keys. We can take an array of size 100 to store the information.

Key	Array of Student information
Key 0 -----> [0]	0 Ramen
Key 1 -----> [1]	1 Raja
Key 2 -----> [2]	2 Bishnu
Key 3 -----> [3]	3 Rajib
Key 4 -----> [4]	4 Mohan
Key 5 -----> [5]	5 Punam
Key 6 -----> [6]	6 Rupshree
....	....
....	....
Key 0 -----> [0]	98 Nilima
Key 0 -----> [0]	99 Amit

Here, we can directly access any student information through the key index because the key index and students roll is the same. This method of searching is called direct addressing. It is useful when the set of possible keys is very small.

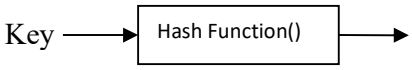
Let us consider the case where we need to store 100 students' information and a five digit roll_no taken as a key index. If we consider the direct addressing method, the key index will be in the range of 00000 to 99999 and the array size will be 100000.

Key	Array of Student information
Key 00000 -----> [0]	00000 Ramen
Key 00001 -----> [1]	00001 Raja
Key 00002 -----> [2]	00002 Bishnu
Key 00003 -----> [3]	00003 Rajib
Key0000 4 -----> [4]	00004 Mohan
Key0000 5 -----> [5]	00005 Punam
Key 00006 -----> [6]	00006 Rupshree
....	....
....	....
Key 99998 -----> [99998]	99998 Nilima
Key 99999 -----> [99999]	99999 Amit

Space for learners:

But in this method we will use 100 locations of the array. Excluding 100 locations, all locations of the array will be unused, which means wastage of space. That is why direct addressing is rarely used.

Now let us discuss how this technique can be improved so that there is no wastage of memory. Here, we will adopt some approach through which we can convert the key within a range. This value will be used as a key index. There are lots of techniques by which we can reduce the size of the array. For example, we can use the last two digits of the key to identify a student. If we use this technique, then the student's information bearing the roll no 54567 is stored in array index 67. Similarly, the student's information bearing the roll no 45859 is stored in the array index 59. This process of converting a key to array index is called hashing and this conversion can be done through hash function.



4.4 HASH TABLE

Hash table is a data structure in which a key is mapped to array locations by a hash function. In simple words, a hash table is an array in which insertion and searching is done through hashing. Hash table stores some elements which basically consist of two main components, i.e., key and value. Key is a unique integer used for indexing the values, whereas Value means data which is associated with key.

4.5 HASH FUNCTION

Hash function is a function which is applied on a key by which it generates an integer within some suitable range in order to reduce the collision that is used as an address of the hash table. Integer returned by the hash function is called hash key. Practically there is no such type of hash function which can fully eliminate collision. But by using a good hash function it can minimize the collision.

Properties of good hash function:

Space for learners:

1. The hash function should generate different hash values for the similar string.
2. The hash function should be easy to compute.
3. The hash function should distribute the keys as uniformly as possible over an array.
4. The hash function should generate the address with a minimum number of collisions.
5. The hash function is a perfect hash function when it uses all the input data.

Space for learners:

4.6 TYPES OF HASH FUNCTION

Here we will discuss some hash function which uses numeric keys. In the real world, sometimes we use alphanumeric key.

4.6.1 Division Method (Modulo-Division)

Here, the key 'x' is divided by the table size 'm' and the remainder will be considered as the address for the hash table. This method ensures that we will get the address in the limited range of the table i.e. we will get the address in the range of {0,1,2……, m-1}. The hash function can be given as

h(x)= x mod m

In this method collision can be minimized if the m value is taken to be prime number.

For example-

For key 'x' = 545 and m = 17,

h(x) = 545 % 17=1 // in c language ' %' is modulo operator

For key = 78549 and m=17

h(x) = 78549 % 17= 9

For key = 59467 and m=17

h(x) = 59467 % 17 = 1

From the above example we can observe that after considering the prime number there are also few chances for collision. This can be

improved by considering a prime number not too close to an exact power of 2 for table_size.

Space for learners:

4.6.2 Multiplication Method

In multiplication method we compute the hash value in four steps:

1. Choose a constant A between 0 and 1.
2. Multiply the key k with A
3. Take the fractional part
4. Multiply the fractional part with m, and take the floor of the result.

Hence the hash function can be written as

$h(x) = \lfloor m((k \times A) \bmod 1 \rfloor$

Where $(k \times A) \bmod 1$ gives the fractional part of kA and m is the total number of indices in the hash table.

Note: This algorithm works better if we choose some values depending on the characteristics being hashed. An American Scientist "Knuth" has suggested that the best choice of A is (sqrt 5-1)/2=0.6180339887.

For example:

Suppose, Hash table of size is 1000

key= 24561

$h(24561) = \lfloor 1000(24561*0.6180339887 \bmod 1 \rfloor$

$= 1000\lfloor 15179.5327964 \bmod 1) \rfloor$

$= 1000* 0.5327964$

$= \lfloor 532.7964 \rfloor$

$= 532$

4.6.3 Mid Square Method

In the mid square method, we need to square the value of the key and take some digits or bits from the middle of this square as the address. This technique can generate keys with high randomness if we take a big enough value. It has some limitations. As the value is

squared, if a 6-digit number is taken, then the square will have 12-digits. This exceeds the range of int data type. So, overflow must be taken care of. if the key is too large then we can take part of the key and perform the mid square method on that part rather than the whole key. The chances of a collision in mid-square hashing are low, not obsolete.

Suppose our keys are four digits integer and table size is 1000. So, we will need a 3 digits address. Now we will square the keys and take the 3rd, 4th and 5th digits value from each squared number as a hash address.

key:	1562	1232	1355	1656
Square of key:	2439844	1517824	1836025	2742336
Address:	398	178	360	423

4.7 COLLISION IN HASH TABLE

Suppose we generated two addresses of an array from different keys using a hash function. If both the addresses of the array generated by the hash function are the same, then this situation is called Collision in Hash table.

Let us describe collision by using an example.

Suppose we need to store Employee information through Emp_id. Here we have considered a hash function that maps a key to the array address by summing up all the digits exists in Emp_id. If the Emp id is 62865, then this employee information will be stored in the array location 27.

Key	Employee information
.....	.....
.....	.....
24	Emp_id 34764
25.....	.....
26	Emp_id 56357
27	Emp_id 62865
28	Emp_id 75691
29	.....
.....	.....

Space for learners:

Now, we would like to store the Employee information of Emp Id 59823. If we add all the digits of Emp_id, the sum is 27. But the location 27 has already been occupied by the Emp_id 62865. This situation is Collision in the hash table. The keys which are mapped to the same address are called synonyms.

Space for learners:

4.8 COLLISION RESOLUTION

A good hash function performs one to one mapping between a set of all possible keys, but it is totally impossible. A collision occurs when a hash function maps two different keys in the same location of a hash table. So, we can use different collision resolution techniques by which these keys can be placed in an alternate location.

Two most important collision resolution techniques we will study here,

i) Open Addressing(Closed hashing)

ii) Separate Chaining(Open Hashing)

4.8.1 Open Addressing (Closed Hashing)

When we map a key and get a particular location in the hash table by using hash function, if the location is already occupied then we will search some other empty location in the hash table and insert the value on it. This method is also known as Closed Hashing because the array is assumed to be closed.

Note: The whole process of examining memory locations in the hash table is called probing. We will mainly study three methods to search for an empty location inside the table when we face a collision.

1. Linear Probing
2. Quadratic Probing
3. Double hashing

4.8.1.1 Linear Probing

When a hash function gives an address which is already occupied in the hash table, In such a case, the hash function searches linearly for

the next empty cell in the hash table. For example, if a hash function gives an address 'a' and suppose it is not empty, then it will search for the next empty location i.e 'a+1'. If this location is also occupied, it will search the next location i.e 'a+2' and this procedure will continue till it finds an empty location where the key can be inserted. During search for an empty position in the array, we assume the array is closed or circular. For example, if we need to store a key value in position 4 , where table size is 10, then we need to search the empty location in the sequence- 4, 5, 6, 7, 8, 9, 0, 1, 2, 3. If the location 4 is empty, we will insert the value in position 4, otherwise we will search linearly for an empty position and store the key value on that position.

Space for learners:

Advantage-

- It is very easy to compute.

Disadvantage-

- The main disadvantage in linear probing is clustering.
- Many consecutive elements form groups.
- Then, it takes time to search for an element or to find an empty bucket.

Time Complexity-

Worst time to search an element in linear probing is O (table size).

4.8.1.2 Quadratic Probing

In linear probing, the main disadvantage is clustering. In quadratic probing this problem is solved by storing the colliding keys away from the initial collision point.

$$H(k,i)=(h(k)+i^2) \bmod Table\_size$$

Here, i varies from 0 to tablesize-1 and h is the hash function. Here also the array is assumed to be closed. The search for empty locations will be in the sequence:

h(k), h(k)+1, h(k)+4, h(k)+9…….. all mod Table_size.

For example,

Table size=10

h(key)=key%10

Key to be inserted 48,34,29,68,98,54,53,74

h(48)= 48%10= 8

h(34)= 34%10= 4

h(29)= 29%10=9

h(68)= 68%10=8

h(98)= 98%10=8

h(54)= 54%10=4

h(53)= 53%10=3

h(74)= 74%10=4

Keys 48,34,29 are inserted without collision. But for the key 68 outcome address is 8, which was already occupied earlier. In such cases we need to search the next free location by the hash function. So the next location will be (8+1)%10=9, which is also occupied. So the nextlocation will be $(8+2^2)$%10=12%10= 2, now it is empty and insert the key at position 2.

Advantages:

Quadratic probing may be a smaller amount likely to possess the matter of primary clustering and is less complicated to implement than Double Hashing.

Disadvantages:

- Quadratic probing has secondary clustering. This happens when 2 keys hash to the identical location, they have the identical probe sequence. So, it takes many attempts before an insertion is being made.
- Also probe sequences don't probe all locations within the table.

4.8.1.3 Double Hashing

In double hashing we will use two independent hash functions rather than a single hash function. Hence, it is called double hashing. The double hash function can be defined as:

$h(k, i) = [h_1(k)+ ih_2(k)]$mod t

Space for learners:

Here, t is the table size, $h_1(k)$ and $h_2(k)$ two independent hash functions where, $h_1(k)$= k mod t and $h_2(k)$= k mod t', where t' will be less than t. and i is probe number start from 0 to t-1.

For example:

Table size(t)= 10, Inserted keys are 54,97,43,27,34

$h_1(k)$= (k mod 10) and $h_2(k)$=(k mod 8)

Initially, the hash table will be as:

We have,

h(k, i) = $[h_1(k)+ ih_2(k)]$mod **t**

Step 1: Key= 54

h(54,0)= [54 mod 10+(0× 54 mod 8)] mod 10

= [4+ (0×0)] mod 10

= [4 mod 10]

=4

(Since position 4 is empty, we will put 54 in location 4.)

0	1	2	3	4	5	6	7	8	9
				54					

Step 2: Key= 97

h(97,0)= [97 mod 10+(0× 97 mod 8)] mod 10

= [6+ (0 * 1)] mod 10

= [6 + 0] mod 10

= 6

(Since position 6 is empty, we will put 97 in location 6)

0	1	2	3	4	5	6	7	8	9
				54		97			

Step 3: Key= 43

h(43 ,0)= [43 mod 10+(0× 43 mod 8)] mod 10

= [3+ (0×3)] mod 10

Space for learners:

= [3+ 0] mod 10

=3

(Since position 3 is empty, we will put 43 in location 3.)

0	1	2	3	4	5	6	7	8	9
				43	54		97		

Step 4: Key=27

h(27 ,0)= [27 mod 10+(0✕ 27 mod 8)] mod 10

= [7+ (0✕3)] mod 10

= [7+ 0] mod 10

=7

(Since position 7 is empty, we will put 27 in location 7)

0	1	2	3	4	5	6	7	8	9
			43	54		97	27		

Step 5: Key= 34

h(34 ,0)= [34 mod 10+(0✕ 34 mod 8)] mod 10

= [4+ (0✕ 2)] mod 10

= [4+ 0] mod 10

=4

0	1	2	3	4	5	6	7	8	9
			43	54		97	27		

(Location 4 is already occupied by the key 54. So we cannot store the key 34 in location 4. We need to find the next location by taking the probe i=1, this time.

h(34 ,1)= [34 mod 10+(1✕ 34 mod 8)] mod 10

= [4+ (1✕ 2)] mod 10

Space for learners:

= [4+ 2] mod 10

= 6 mod 10

=6

0	1	2	3	4	5	6	7	8	9
			43	54		97	27		

(Location 6 is again occupied by the key 97. So we cannot store the key 34 in location 6. We need to find the next location by taking the probe i=2, this time.

h(34 ,2)= [34 mod 10+(2✕ 34 mod 8)] mod 10

= [4+ (2✕ 2)] mod 10

= [4+ 4] mod 10

= 8 mod 10

=8

0	1	2	3	4	5	6	7	8	9
			43	54		97	27	34	

Now the location 8 is empty, we will put 34 in location 8.

We will repeat the entire process by increasing the probe by 1 until we will not get the empty location.

Advantages:

- Double hashing eliminates the problems of the clustering issue.

Disadvantages:

- Double hashing is more complicated to implement than any other hashing.
- Double hashing can cause thrashing.

Space for learners:

4.8.2 Separate Chaining (Open Hashing)

Space for learners:

In this technique, when we face the problem of the same hash address a Linked list are maintained for those elements. Here the hash table does not contain actual keys and records but it is just an array of pointers, where each pointer points to a linked list. That is location 1 in the hash table points to the head of the linked list of all the key values that is hashed to 1. If there is no key value hashes to 1, the location is set to NULL.

For example:

The Keys are 267, 341,223,674, 755, 921,733,874, 231,397 which are to hashed

Table size= 10

h(267)=267 mod 10= 7

h(341)=341 mod 10 = 1

h(223)= 223 mod 10= 3

h(674)=674 mod 10 = 4

h(755)=755 mod 10= 5

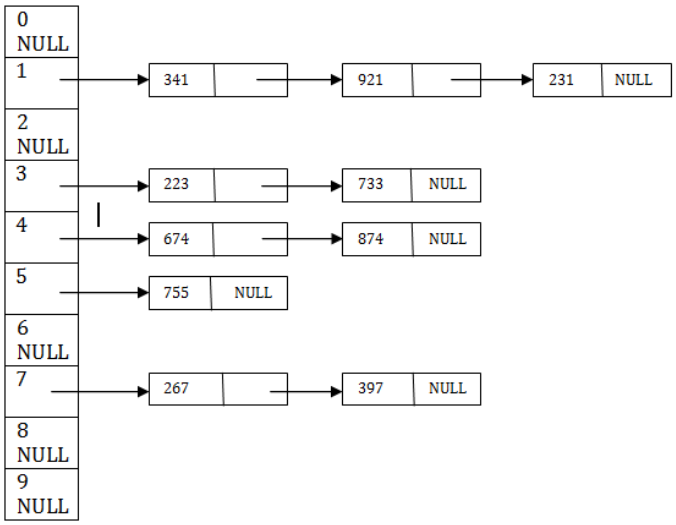
h(921)= 921 mod 10=1

h(733)= 733 mod 10= 3

h(874)=874 mod 10=4

h(231)= 231 mod 10= 1

h(397)=397 mod 10=7



Advantages:

- Not depending on the size of the table.
- Implementation is very simple.

Disadvantages:

- Keys are not evenly distributed in separate chaining.
- Misuse of space due to lots of empty spaces in the table.
- The list in the positions can be very long.

Space for learners:

CHECK YOUR PROGRESS

A. Multiple Choice Question and Answer:

1. The searching technique that takes O (1) time to find a data is
   a) Linear Search
   b) Hashing
   c) Binary Search
   d) Tree Search
2. What is a hash table?
   a) A structure that maps keys to values
   b)A structure that maps values to keys
   c) A structure used for storage
   d)A structure used to implement stack and queue

3. Which Open addressing technique is free from clustering problem?
   a) Linear Probing
   b) Quadratic probing
   c) Double hashing
   d) None of the above
4. What is direct addressing?
   a) Distinct array position for every possible key
   b) Fewer array positions than keys
   c) Fewer keys than array positions
   d) Same array position for all keys
5. What is a hash function?
   a) A function has allocated memory to keys
   b) A function that computes the location of the key in the array
   c) A function that creates an array
   d) A function that computes the location of the values in the array.
6. In separate chaining, which data structure is normally used?
   a) Singly linked list
   b) Doubly linked list
   c) Circular linked list
   d) Binary trees
7. Which hash function used in Division method?
   a) h(k)= k/m, where k is the key and m is the table size.
   b) h(k)= k mod m, where k is the key and m is the table size
   c) h(k)= m/k, where k is the key and m is the table size
   d) h(k)= m mod k, where k is the key and m is the table size

B. Fill up the Blanks and Answer:

1. The hash function should generate different _____for the similar string.
2. A collision occurs when a hash function maps _____in the same location of a hash table.
3. The main disadvantage in linear probing is_____.
4. Worst time to search an element in linear probing is _____.
5. Quadratic probing has _____clustering.
6. In Division method, the key is divided by the table size and the _____will be considered as the address for the hash table.

Space for learners:

4.9 SUMMING UP

Space for learners:

- In hashing we get O(1) search time on an average and O(n) in the worst case. Hashing is an improvement over **Direct Access Table.**
- **Hashing** is a technique to convert a range of key values into a range of indexes of an array.
- **Hash table** is a data structure in which a key is mapped to array locations by a hash function. In simple words, a hash table is an array in which insertion and searching is done through hashing.
- Suppose we generated two addresses of an array from different keys using a hash function. If both the addresses of the array generated by the hash function are the same, then this situation is called **Collision** in Hash table.
- **Hash function** is a function which is applied on a key by which it generates an integer within some suitable range in order to reduce the collision that is used as an address of the hash table.
- Most popular Hash function which are **Division Method Modulo-Division), Multiplication Method, Mid-Square Method** etc.
- In **Division method,** the key is divided by the table size and the remainder will be considered as the address for the hash table.
- In **Multiplication method** applies the hash function as h(x)= ⌊ m((k ✕A) mod 1⌋

  Where (k ✕A) mod 1 gives the fractional part of kA and m is the total number of indices in the hash table.
- In **Mid-Square Method**, Key value is squared and take some digits or bits from the middle of this squared value as the address. This technique can generate keys with high randomness if we take a big enough value.
- A collision occurs when a hash function maps two different keys in the same location of a hash table. Different collision resolution techniques can be adopted by which these keys can be placed in an alternate location.

- Two most important collision resolution techniques we will study here, they are **Open Addressing (Closed hashing)** and **Separate Chaining (Open Hashing).**
- In **Open Addressing**, computes a new position using a probe sequence and the next record is stored in that position.
- Open addressing can be implemented by using three methods named as **Linear Probing, Quadratic Probing and Double hashing.**
- In **Linear Probing**, the hash function searches linearly for the next empty cell in the hash table. For example, if a hash function gives an address 'a' and suppose it is not empty, then it will search for the next empty location i.e 'a+1' and so on.
- In **Quadratic Probing**, if a value is already occupied at a location generated by the h(k) then the following hash function can resolved the problem.

  $H(k,i)=(h(k)+i^2)$ mod Table_size

  Here, i varies from 0 to tablesize-1 and h is the hash function. Here also the array is assumed to be closed. The search for empty locations will be in the sequence:

  h(k), h(k)+1, h(k)+4, h(k)+9…….. all mod Table_size.
- In **Double Hashing**, we will use two hash functions rather than a single function. The double hash function can be defined as:

  $h(k, i) = [h_1(k)+ ih_2(k)]$ mod t

  Here, t is the table size, $h_1(k)$ and $h_2(k)$ two independent hash functions where, $h_1(k)$= k mod t and $h_2(k)$= k mod t', where t' will be less than t. and i is probe number start from 0 to t-1.
- In **Separate chaining** method, each location in as hash table stores a pointer to a linked list that contains all the key values that were hashed to that location.

Space for learners:

4.10 ANSWERS TO CHECK YOUR PROGRESS

A. Answers:

Question Number	Answer Key
1	b) Hashing
2	a) A structure that maps keys to values
3	c) Double hashing
4	a) Distinct array position for every possible key
5	b) function that computes the location of the key in the array
6	(a) Singly linked list
7	(b) h(k)= k mod m, where k is the key and m is the table size

B. Answers:

1. Hash values
2. two different keys
3. clustering
4. O (table size).
5. Secondary
6. remainder

4.11 POSSIBLE QUESTIONS

Short type questions:

1. What is Hashing?
2. What is Hash Table?
3. What is the importance of Hashing?
4. What is collision in Hash Table?
5. Explain the Division Method.
6. Explain Double Hashing.

Long answer type questions:

1. What is hashing? Write few applications of Hashing?
2. Explain different types of Hash function.

Space for learners:

3. What is collision in Hash table? Write a brief overview of different collision Resolution Techniques**.**

Space for learners:

4.12 REFERENCES AND SUGGESTED READINGS

1. Data Structures Through C In Depth. By S.K. Srivastava/Deepali Srivastava.
2. Data Structures Using C by Reema Thareja.

UNIT 5: SORTING ALGORITHMS AND SELECTION TECHNIQUE I

Unit Structure:

Space for learners:

5.1 INTRODUCTION

A sorting algorithm puts elements of a list in a certain order. Here, we will discuss three most powerful sorting algorithms, namely Quick Sort, Heap Sort and Shell sort. Various sorting techniques are already developed. Each sorting technique has some advantages and disadvantages. Suppose if we need a faster algorithm commonly we use Quick Sort. Quick sort is best on unsorted data sets. Heap sort is used when memory usage is a concern. Shell sort does not use extra memory space.

5.2 UNIT OBJECTIVES

The main aim of study various sorting algorithm is for information searching. In this chapter we will study three best searching techniques named as Quick sort, Heap sort and Shell sort. Quick sort is one of the fastest algorithm so it is widely used. We can mention that Quick sort is a cache-friendly algorithm as it has a good locality of reference when used for arrays. By Heap Sort we can find out the smallest (shortest) or highest (longest) value is needed instantly. It is also used to deal with priority queues in Prim's algorithm and Huffman encoding or data compression. Insertion sort does not perform good when the close elements are far apart. Shell sort helps to reduce the distance between the close elements. Shell sort is an optimization of insertion.

Space for learners:

5.3 CLASSIFICATION OF SORTING ALGORITHMS

Internal sorting:

If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

External sorting:

When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory success hard disk, floppy disk, magnetic tapes etc. then external sorting methods are performed. (Natural, Polyphase, Balanced techniques used here)

5.4 QUICK SORT

In 1962 C.A.R Hoare developed the quick Sort algorithm based on divide and conquer technique. In this technique, a problem is subdivided into smaller units which is again sub -divided into smaller units and so on i.e., a big problem is solved in small units. It is in-place sorting problems that's no extra memory is required. Quick sort average case performance is O (nlogn). Worst case performance is O(n2). Quick sort also called partitioned exchange sort.

Now let us explain the sorting procedure. We will choose an element from the list and which will be considered as pivot. Now we need to rearrange the elements in such a way that all the elements to the left of the pivot element are less than to the pivot element and all the elements to the right of the pivot element are greater than the pivot element. If there is any value equal to pivot elements, it can go either way. After such partitioning, the pivot element is placed in the final position. Now recursively sort the two sub lists thus obtained (one with a sublist of elements smaller than pivot element and the other having the greater value elements). Here, after completion of each iteration, one element (pivot) will be in its final position and that's after completion of each iteration one less element to be sorted in the list. Pivot element can be any element from the array, it can be the first element, the last element, any random element or median element.

Space for learners:

5.4.1 Example

Now we will elaborate the whole procedure by considering an example.

The elements which need to be sorted are 10, 8, 14, 7, 12, 5, 3, 17, 10.

PIVOT=10

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
10	8	14	7	12	5	3	17	10

Start **End**

First check the pivot elements with the start element. If the pivot element is less than or equal to the start element, increment the position of start. Here, start value 10 is equal to pivot, so increment the position of start. Now start at array index 1.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Space for learners:

10	8	14	7	12	5	3		17	10

Start

End

Now compare starting elements 8 with pivot 10. It is less than pivot. So increment the position of start. Now start at array index 2.

0	1	2	3	4	5	6	7	8
10	8	14	7	12	5	3		17	10

Start

End

Now the start value is greater than pivot, so will stop the increment. So, we will now move to the end position. Now check the end value with pivot. If the end value is greater than the pivot, we need to decrement the end position. Here the end value is 10, which is equal to the pivot. In this condition we will swap the end value with the start value.

0	1	2	3	4	5	6	7	8
10	8	10	7	12	5	3		17	14

Start

End

Again we compare the start value with pivot. Here the start value is equal to pivot so increment the position of start. Now start at the array index 3.

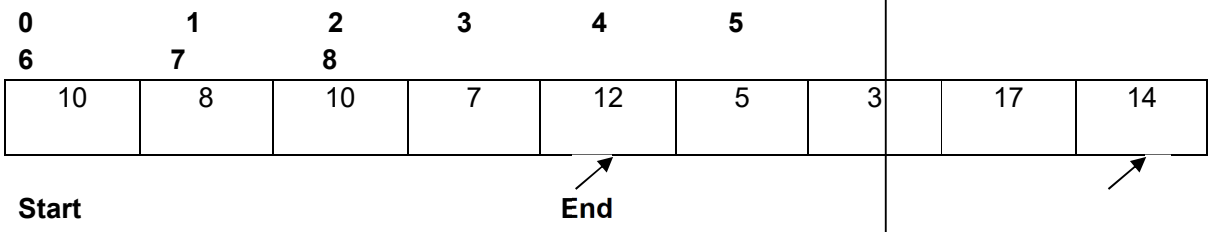
0	1	2	3	4	5	6	7	8
10	8	10	7	12	5	3		17	14

Start

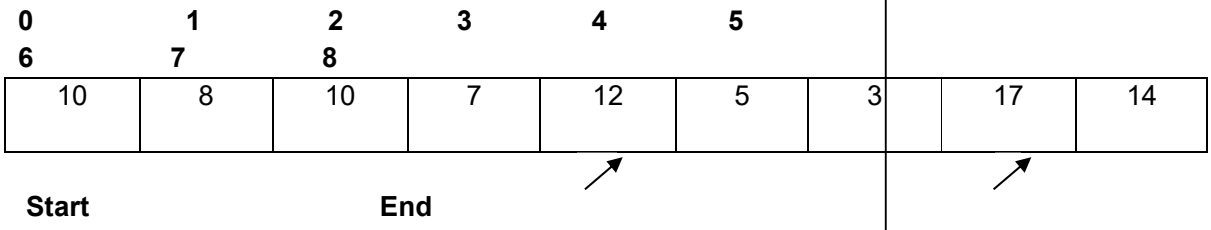
End

Here, again we will compare the start value with pivot. Here the start value is less than pivot so increment the position of start. Now start at the array index 4 .

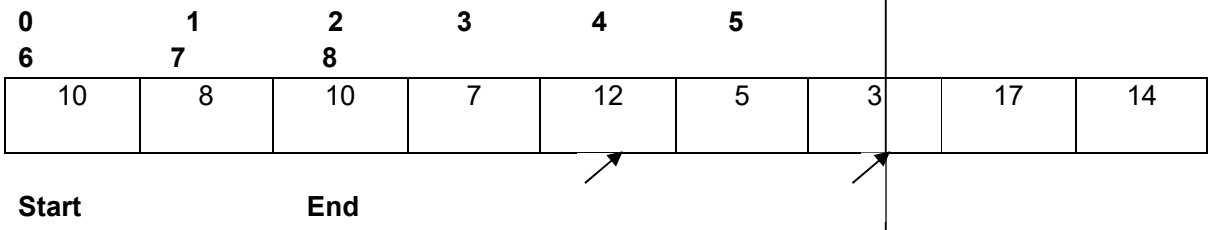
Space for learners:



We will compare the start element 12 with pivot element 10. Start value is greater than pivot element 10. So, we need to stop here.
Now we will move to the end position. We will compare the end value 14 with pivot element 10. Here 14 is greater than 10. So, decrement the position of End.



Now, again check the 17 with pivot value 10, 17 is greater than pivot 10, so decrement the end position.



Now, again check the end value 3 with pivot value 10, 3 is not greater than pivot 10, so will stop here. After that we will swap start value with end value.

Space for learners:

0	1	2	3	4	5	6	7	8
10	8	10	7	3	5	12	17	14

Start **End**

Again we will move to the start position and compare the start value 3 with pivot 10. It is less than pivot, so increment the position of start.

0	1	2	3	4	5	6	7	8
10	8	10	7	3	5	12	17	14

Start **End**

Now compare the start value with pivot 10, it is less than pivot, so increment the start position.

0	1	2	3	4	5	6	7	8
10	8	10	7	3	5	12	17	14

Start End

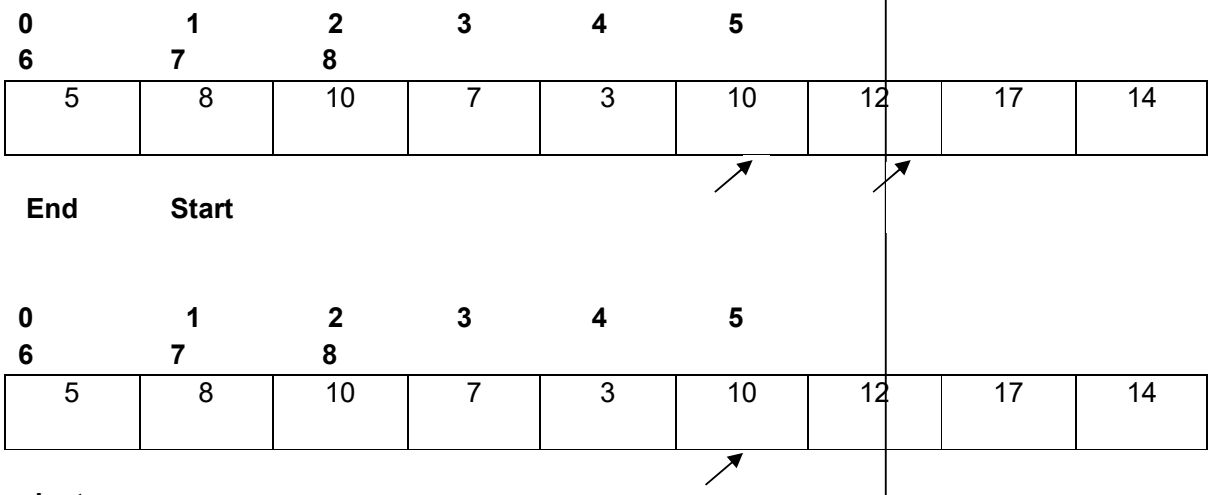
Now compare the start value with pivot 10, it is not less than pivot, so we will stop here. Now we will move to the end position and compare with pivot. Here we see that the end value is greater than pivot. So decrement the position of end.

0	1	2	3	4	5	6	7	8
10	8	10	7	3	5	12	17	14

End **Start**

Space for learners:

Now, upto this position we follow where start is less than end, But here end is less than start, so we will not swap the start value with end value. In this situation we will swap pivot element with end value.



Now, here we can observe that all the elements in the left side of pivot element have lesser value than pivot and all the elements in the right side of pivot have greater value than pivot.

Now, the pivot element is now placed in its proper position. Now we will subdivide the list in two parts. One is array index 0 to 4 and another one is 6 to 8. Again we apply the same procedure on those two parts by considering pivot elements.

5.4.2 Functions related to Quick sort

Some necessary functions of quick sort have been given below-

```
void  swap(int *a, int *b)
{
int t;
```

```
        t= *a;
        *a=*b
        *b=t;
}

void quicksort (int  a[ ], int p, int r )  //a[ ]  is the array, p is starting
index of the array,
and r is the last index of array
{
if (p <r )
        {
        int q;
                q= partition (a, p, r)
                quicksort (a, p, r);
                quicksort (a, q+1,r);
        }
}

int partition(int a [ ], int low, int high)
{
int i;
int pivot=arr[high];   // selecting last element as pivot,
                              //we also can take the first element as
                          pivot


      i=(low-1) // index of smaller element
for(j=low; j<= high-1; j++)
       {
if(arr[j]<=pivot)    // if current element is smaller than or equal to
pivot
           {
```

Space for learners:

```
            i++;
swap(&arr[i], &arr[j]);
             }
}
swap(&arr[i+1], &arr[high]);
return(i+1);
}

//function to print the array
voidprintArray(int a[ ], int size]
{  int i;
for(i=0;i<size; i++)
{   print("%d", a[i]);
}
}
```

5.4.3 Analysis of Quick sort

Best case time complexity of quick sort is O(n log(n)) and we will get it when we will select pivot as a mean element. Worst case time complexity of quick sort is O(n^2) and we will get it when our array will be sorted and we select smallest or largest indexed element as pivot.

Average case time complexity of quick sort is O(n log(n))

5.5 HEAP SORT

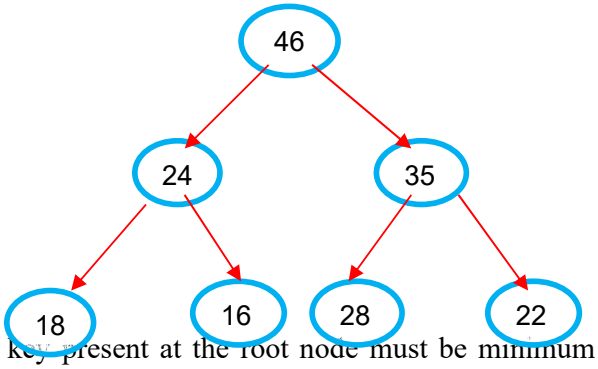
A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Heap sort is one of the best sorting methods being In-place and with no quadratic worst case running time.

Space for learners:

Generally, Heap can be of two types-

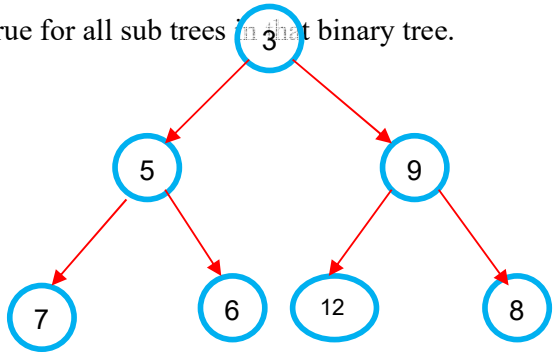
5.5.1 Max- Heap

In a Max Heap, the key presents at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.



5.5.2 Min-Heap

In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub trees in that binary tree.



In heap sort there are only two phases:

1. First, start with the construction of a heap(Max-Heap or Min-Heap) from the array elements.
2. After creation of Heap, keep on eliminate the root element of the heap by shifting it to the end of the array and store the heap

Space for learners:

structure with remaining elements till there is only one element in the tree.

Space for learners:

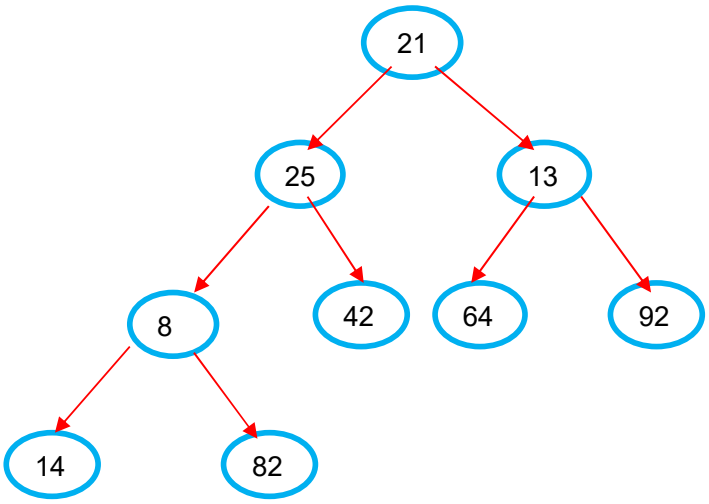
5.5.3 Example

Let us explain the working principal of heap sort with an example.

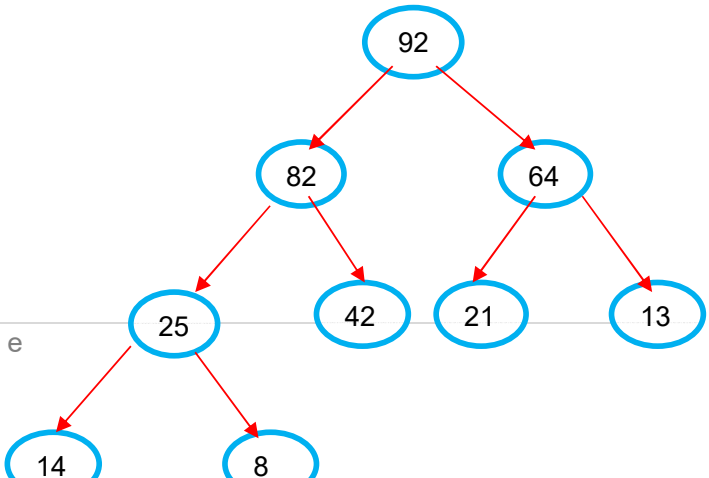
The elements are to be sorted are

21 25 13 8 42 64 92 14 82

First we will build a simple binary tree on the above element.



Now, make the max heap on the above binary tree.



Space for learners:

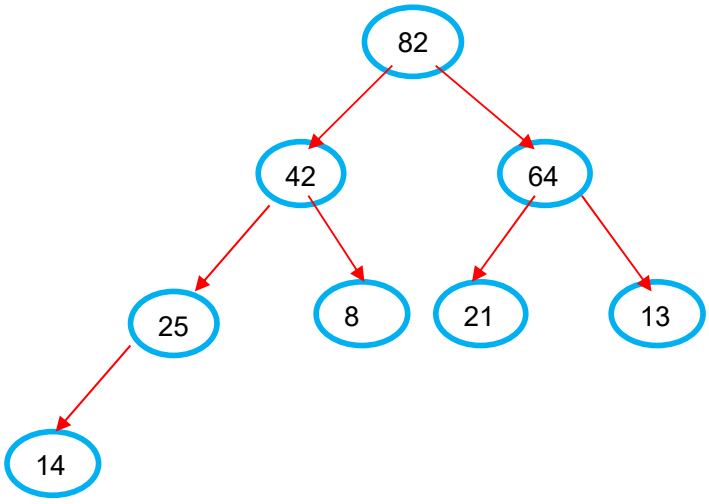
Now the array is

0	1	2	3	4	5	6	7	8
92	82	64	25	42	21	13	14	8

Now swap the value 0^{th} position with last leaf node value.

0	1	2	3	4	5	6	7	8
8	82	64	25	42	21	13	14	92

Make the max heap with the above array element excluding the last root value.



Now the array is

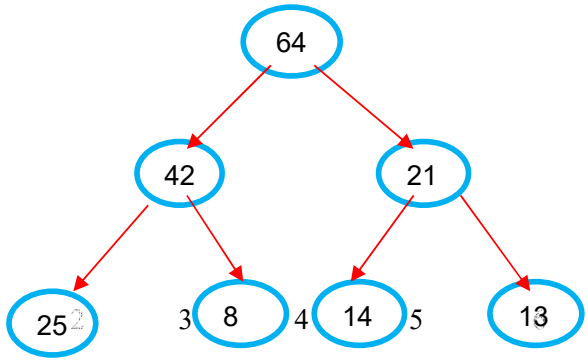
0	1	2	3	4	5	6	7	8

82	42	64	25	8	21	13	14	**92**

Now swap the value 0th position with last leaf node value.

0	1	2	3	4	5	6	7	8
14	42	64	25	8	21	13	**82**	**92**

Make the max heap with the above array element excluding the last root value.



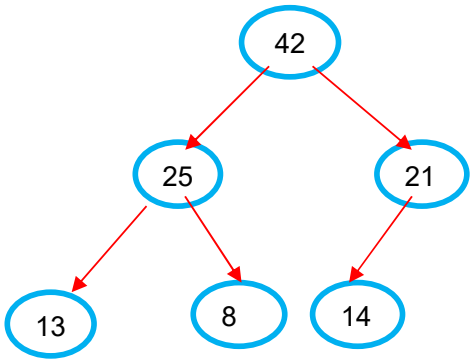
Now the array is

0	1	2	3	4	5	6	7	8
64	42	21	25	8	14	13	**82**	**92**

Now swap the value 0th position with last leaf node value.

0	1	2	3	4	5	6	7	8
13	42	64	25	8	21	**64**	**82**	**92**

Make the max heap with the above array element excluding the last root value.



Now the array is

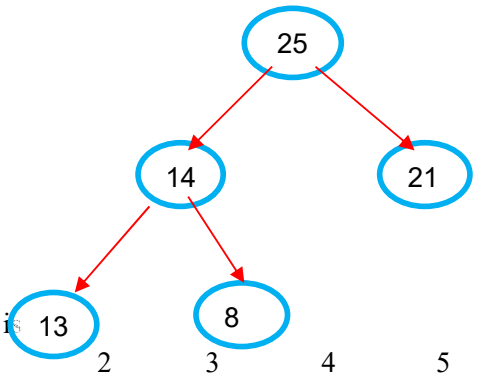
Space for learners:

0	1	2	3	4	5	6	7	8
42	25	21	13	8	14	**64**	**82**	**92**

Now swap the value 0th position with last leaf node value.

0	1	2	3	4	5	6	7	8
14	25	21	13	8	**42**	**64**	**82**	**92**

Make the max heap with the above array element excluding the last root value.



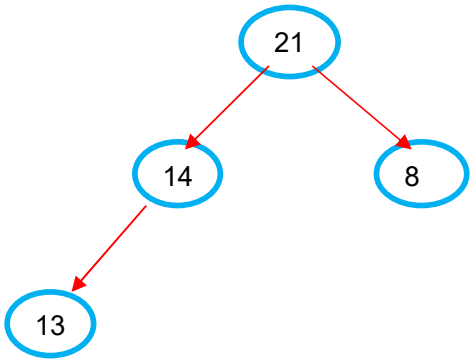
Now the array is

0	1	2	3	4	5	6	7	8
25	14	21	13	8	**42**	**64**	**82**	**92**

Now swap the value 0th position with last leaf node value.

0	1	2	3	4	5	6	7	8
8	14	21	13	**25**	**42**	**64**	**82**	**92**

Make the max heap with the above array element excluding the last root value.



Space for learners:

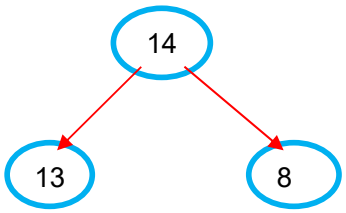
Now the array is

0	1	2	3	4	5	6	7	8
21	14	8	13	**25**	**42**	**64**	**82**	**92**

Now swap the value 0^{th} position with last leaf node value.

0	1	2	3	4	5	6	7	8
13	14	8	**21**	**25**	**42**	**64**	**82**	**92**

Make the max heap with the above array element excluding the last root value.



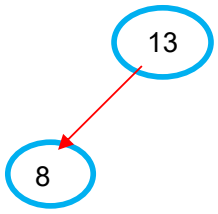
Now array is

0	1	2	3	4	5	6	7	8
14	13	8	**21**	**25**	**42**	**64**	**82**	**92**

Now swap the value 0^{th} position with last leaf node value.

0	1	2	3	4	5	6	7	8
8	13	**14**	**21**	**25**	**42**	**64**	**82**	**92**

Make the max heap with the above array element excluding the last root value.



Now array is

Space for learners:

0	1	2	3	4	5	6	7	8
13	8	**14**	**21**	**25**	**42**	**64**	**82**	**92**

Now swap the value 0^{th} position with last leaf node value.

0	1	2	3	4	5	6	7	8
8	**13**	**14**	**21**	**25**	**42**	**64**	**82**	**92**

Now we have only one element to make the heap. So it will remain in the same position.
The Final sorted array is

0	1	2	3	4	5	6	7	8
8	**13**	**14**	**21**	**25**	**42**	**64**	**82**	**92**

5.5.4. Functions related to Heap sort

Some basic function used in heap sort-

```
voidheapify (intarr[ ], int n, int l)
{       int largest, l,r;
        largest=i;
        l= 2*i+1;
        r=2*i+2;
        //if left child is larger than root
        if(l<n &&arr[l]>arr[largest])
        largest=l;
        // if right child is larger than largest so far
        if( r<n &&arr[r]>arr[largest]
        largest=r;
        //if largest is not root
        if(largest !=i)
        { swap(arr[i], arr[largest]);
```

Space for learners:

```
        //recursively heapify the affected sub-tree
        heapify(arr, n, largest);
        }
        }

        voidheapSort(intarr[ ], int n)
        {    int i;
          // build heap
        for(i=n/2-1; i>=0; i--)

        heapify(arr, n, i)
           // one by one extract an element from heap
        for(i=n-1; i>=0; i--)
          {
        //move current root to end
                swap(arr[0],arr[i]);
                //call max heapify on the reduced heap
                heapify(arr, i, 0 );
          }
}
```

5.5.5 Analysis of Heap sort

We willanalyze the heap sort in two phases. For building a heap its running time is O(n), if we use bottom up approach. For delete operation in a heap takes O(log n) and it is called n-1 times, so the complexity in this phase is O(n log n). So the worst case analysis of heap sort is O(n log n). The average and best case complexity is also O(n log n).

5.6 SHELL SORT

Space for learners:

Shell sort was developed by Donald Shell in the year 1959. Shell sort is just an improvement of insertion sort. It is in-place comparison sort as it requires no additional space. In insertion sort we have observed that it gives better performance when input data is almost sorted. Another disadvantage of insertion sort is that it is inefficient because many moves are performed and it moves just one position at a time. The method starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Here, elements are sorted in multiple passes and in each pass; data are taken with smaller and smaller gap sizes. And when we reach the last step, we implement the insertion sort technique on those data. In the last step of shell sort, the elements are already sorted, and already we know that performance of insertion sort will be better when data is almost sorted, hence it provides good performance.

Space for learners:

5.6.1 Example:

Here we will visualize the way of shell sort by taking an example.

The elements are:

49, 29, 38, 17, 14, 16, 24, 43

Original list	**49**	**29**	**38**	**17**	**14**	**16**	**24**	**43**
Pass 1: **Gap =floor(n/2)** **= floor(8/2)** **= 4**	**49**	**__**	**__**	**__**	**14**	**__**	**__**	**__**
	__	**29**	**__**	**__**	**__**	**16**	**__**	**__**
	__	**__**	**38**	**__**	**__**	**__**	**24**	**__**
	__	**__**	**__**	**17**	**__**	**__**	**__**	**43**

Sort on sublist	**14 __ __ __ 49 __ __ __** **__ 16 __ __ __ 29 __ __** **__ __ 24 __ __ __ 38 __** **__ __ __ 17 __ __ __ 43**	***Space for learners:***
Combine this sorted sublist	**14 16 24 17 49 29 38 43**	
Pass 2: **Gap = floor(4/2)** **= 2**	**14 __ 24 __ 49 __ 38 __** **__ 16 __ 17 __ 29 __ 43**	
Sort on sublist	**14 __ 24 __ 38 __ 49 __** **__ 16 __ 17 __ 29 __ 43**	
Combine this sorted sublist	**14 16 24 17 38 29 49 43**	
Pass 3: **Gap = floor(2/2)** **= 1** **(Here we will follow insertion sort technique)**	**14 16 24 17 38 29 49 43**	
Sort on sublist	**14 16 17 24 29 38 49 43**	

5.6.2 Functions related to Shell sort

The function of shell sort:

```
voidshellSort( int array[ ], int n)
{ int gap, i,j, temp;
for (gap=n/2; gap>0; gap/=2)
{
```

```
        for (i=gap;i<n; i+=1)
        {
                temp= array[ i];
                for( j=i; j>=gap && array[j-gap]> temp; j-=gap)
                {
                array[j]=array[j-gap];
                }
                array[j]=temp;
        }
   }
}
```

Space for learners:

5.6.3 Analysis of Shell Sort

Shell Sort is a comparison based sorting. Time complexity of Shell Sort depends on gap sequence. Its best case time complexity is O(n * log n). Worst case is $O(n*\log^2 n)$. The time complexity of Shell sort is generally assumed to be near to O(n) and less than $O(n^2)$ as determining its time complexity steel and open problem.

The best case in shellsort is when the array is already sorted. The number of comparisons is less. It is an in-place sorting algorithm as it requires no additional scratch space. Shell sort is unstable sort as the relative order of elements with equal values may change.

Check Your Progress:

Multiple Choice Question:

I. The time complexity of a quick sort algorithm which makes use of median, found by an O(n) algorithm, as pivot element is
   a) $O(n^2)$
   b) O(nlogn)

c) O(nloglogn)

d) O(n)

II. Which of the following is not a non comparison sort?

a) Counting sort

b) Bucket sort

c) Radix sort

d) Shell sort

III. The time complexity of heap sort in worst case is

a) O(logn)

b) O(n)

c) O(nlogn)

d) $O(n^2)$

IV. Which of the following algorithms has lowest worst case time complexity?

a) Insertion sort

b) Selection sort

c) Quick sort

d) Heap sort

V. Which of the following algorithm design technique is used in the quick sort algorithm?

a) Dynamic programming

b) Backtracking

c) Divide-and-conquer

d) Greedy method

Space for learners:

5.7 SUMMING UP

- Various sorting techniques are already developed. Each sorting technique has some advantages and disadvantages. Suppose if we need a faster algorithm commonly we use quick sort. Quick sort is best on unsorted data sets. Heap sort is used when memory usage is a concern. Shell sort does not use extra memory space.

- If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

- When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be keep in auxiliary memory success hard disk, floppy disk, magnetic tapes etc. , then external sorting methods are performed.

- The quick Sort algorithms follow the divide and conquer technique. In this technique, a problem is subdivided into smaller units which are again sub -divided into smaller units and so on i.e. a big problem is solved in small units.
- Best case time complexity of quick sort is O(n log(n)) and we will get it when we will select pivot as a mean element. Worst case time complexity of quick sort is O(n^2) and we will get it when our array will be sorted and we select smallest or largest indexed element as pivot. Average case time complexity of quick sort is O(n log(n))
- A heap is a special Tree-based data structure in which the tree is a complete binary tree. Heap sort is one of the best sorting methods being In-place and with no quadratic Worst case running time. Generally
- Generally heap can be of two types: Max Heap and Min Heap.

Space for learners:

- In a Max Heap, the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.
- In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub trees in that binary tree.
- In Heap sort, first, start with the construction of a heap (Max-Heap or Min-Heap) from the array elements. After creation of Heap, keep on eliminate the root element of the heap by shifting it to the end of the array and store the heap structure with remaining elements till there is only one element in the tree.
- Analysis of Heap sort can be done in two phases. For building a heap its running time is O(n), if we use bottom up approach. For delete operation in a heap takes O(log n) and it is called n-1 times, so the complexity in this phase is O(n log n). So the worst case analysis of heap sort is O(n log n). The average and best case complexity is also O(n log n).
- In Shell sort, the method starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Here, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes. When we reach the last step, we implement the insertion sort technique on those data.
- Time complexity of Shell Sort depends on gap sequence. Its best case time complexity is O(n * log n). Worst case is $O(n*\log^2 n)$.

Space for learners:

- The best case in shell sort is when the array is already sorted. The number of comparisons is less. It is an in-place sorting algorithm as it requires no additional scratch space.

Space for learners:

5.8 ANSWERS TO CHECK YOUR PROGRESS

Answer: I. b) O(nlogn)

II. d) Shell sort

III.c) O(nlogn)

IV. d) Heap sort

V. c) Divide-and-conquer

5.9 POSSIBLE QUESTIONS

Short type questions:

I. Define Min heap and Max Heap.

II. Write the working principle of heap sort.

III. Write the working principle of quick Sort.

IV. Write the working principle of shell sort.

V. Write the best case, worst case and average case complexity of Heap sort.

VI. Write the best case, worst case and average case complexity of Quick sort.

VII. Write the best case, Worst case and average case complexity of Shell sort.

Broad type questions:

I. Explain the working principle of shell sort, heap sort and quick sort by consider the following unsorted elements.

34, 23, 5, 18, 39, 67, 45, 76, 46, 29

II. Implement the shell sort, heap sort and quick sort taking at least 50 elements and show their complexity analysis in a single C program.

5.10 REFERENCES AND SUGGESTED READINGS

- Data Structures Through C In Depth. By S.K.Srivastava /Deepali Srivastava..
- Data Structures Using C by Reema Thareja.
- Algorithms, Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest

---×---

Space for learners:

UNIT 6: SORTING ALGORITHMS AND SELECTION TECHNIQUES II

Unit Structure

Space for learners:

6.1 INTRODUCTION

Till now we have seen that all the sorting algorithms are sorted by comparing the input elements in between them. Hence these algorithms referred to as comparison sorting algorithms also. In this chapter we are going to learn two sorting algorithms: counting and radix sort that is based on operations other than comparison. Also

we will analyse to find out that both these algorithms run in linear time. After the sorting algorithms there will be an introductory part of medians and order statistics and finally the selection problem using Randomized Select algorithm.

Space for learners:

6.2 UNIT OBJECTIVES

After going through this unit, you will be able to understand:

- logic behind counting sort
- analysis of the running time of counting sort
- logic behind the radix sort
- analysis of the running time of radix sort
- median and order statistics
- Randomized Select Algorithm

6.3 COUNTING SORT

Counting sort is a sorting algorithm that finds maximum element of the given array and then creates an auxiliary array of the maximum element size. The new auxiliary array consists of the number of occurrences of the distinct elements in the array. The sorting is done by mapping the count as an index of the auxiliary array. Counting sort is generally used when the range of input values isn't greater than the number of values to be sorted. Thus it enables the algorithm to run in linear time that is closer to O(n). The algorithm of counting sort is given below. Here A[1,2…..n] is the given array to be sorted, B[1,2……n] is the output array, thus length[A] = n. C[0,1,….k] is the auxiliary array which works as a temporary storage array.

COUNTING-SORT

1. for i← 0 to k
2. do C[i] ← 0
3. for j ← 1 to length[A]

4. do C[A[j]] ← C[A[j]] + 1

5. Δ C[i] now contains the number of elements equal to i

6. for i ← 1 to k

7. do C[i] ← C[i] + C[i] + 1

8. Δ C[i] now contains the number of elements less than or equal to i

9. for j ← length[A] downto 1

10. do B[C[A[j]]] ← A[j]

11. C[A[j]] ← C[A[j]] – 1

Now let us understand the algorithm step by step.

In lines 1-2 initialization is done.

In for loop of lines 3-4 we examine each input element and if the value of an input element is i, the value of C[i] is incremented.

Thus, after line 4, C[i] will have the number of elements equal to i for each integer i=0,1,…k.

In lines 6-7 for each i = 0,1….k, how many elements are less than or equal to i is determined by keeping the running sum of array C.

In lines 9-11 we place the elements in the correct position of the output array. If all the n elements are distinct then C[A[j]] is the final correct position of A[j]. But the elements may not always be distinct, so we decrement the value of C[A[j]] everytime we place an A[j] in the output array B.

C[A[j]] is decremented so that if the next input element's value is equal to A[j], it is placed in the position immediately before A[j] in the output array.

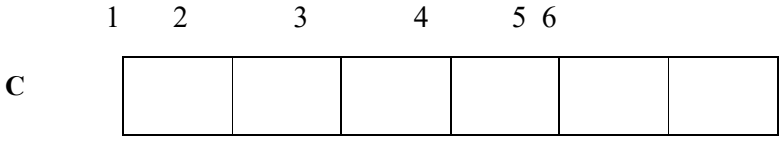
Let us understand the logic of the algorithm with the help of an example given below.

Let A be the input array with eight number of elements

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

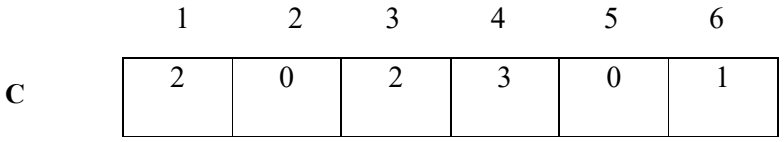
Space for learners:

As we have seen that the elements in the input array ranges from 1 to 6, so when we construct array C, its size will be 6 starting from 0.

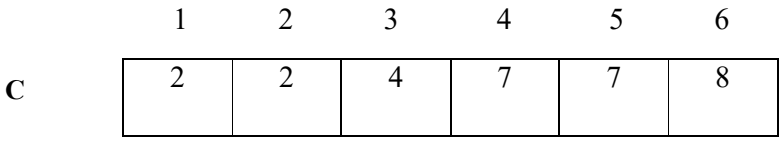


Now we will fill up array C by keeping counts of the corresponding elements in array A

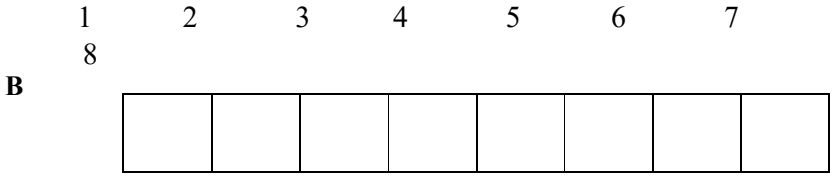
As we can see that the digit 1 occurs two times in array C. Hence the first location of array C i.e.C[1] will consist 2. Similarly if we observe that the number 2 doesn't occur at all, so its count in array C will be zero i.e.C[2] = 0. In this way after filling all the locations, the array C will look like this:



We will modify the elements of the count array C by summing the previous value and the present value and placing it in the present position. The element in the initial position will remain same i.e.C[1] = 2



Now we assume B to be the output array and its dimension will be similar to the input array A.

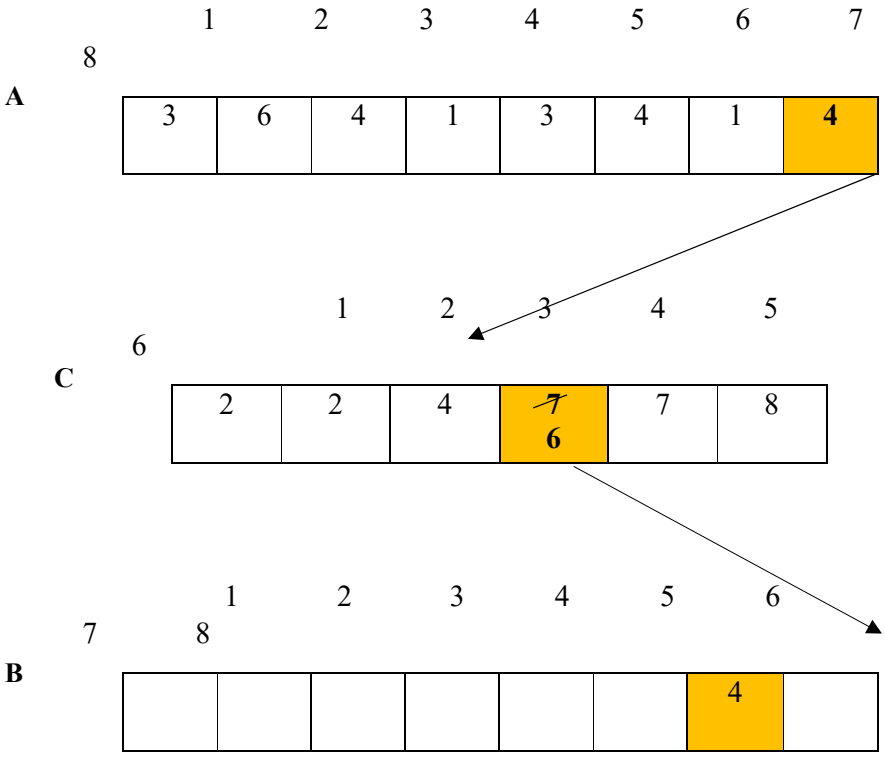


Space for learners:

So finally we will do the sorting by going through array A and then C and finally putting the element in its correct position in array B.

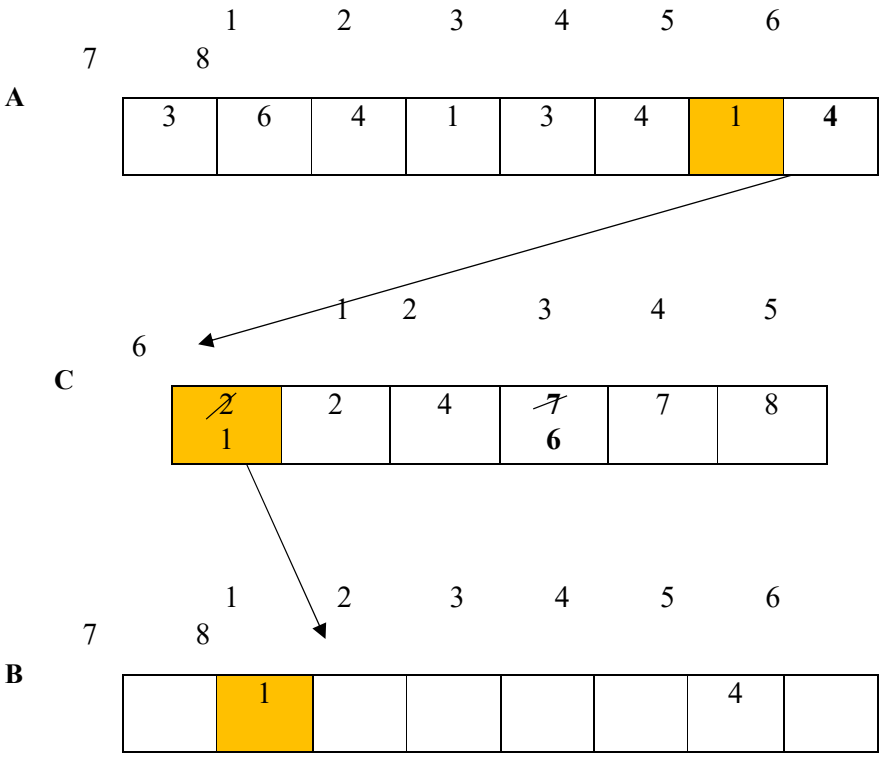
First we will start from the last element of array A i.e. element 4 which is in the location A[8]. The element in A[8] gives the index in array C that we are going to consider.. In this case we get the element 7 in C[4]. Again the element in C[4] gives the index for the output array B Finally 4 will be placed in index 7 of the output array B and decrement the number 7 in array C by 1.

The steps are shown below:

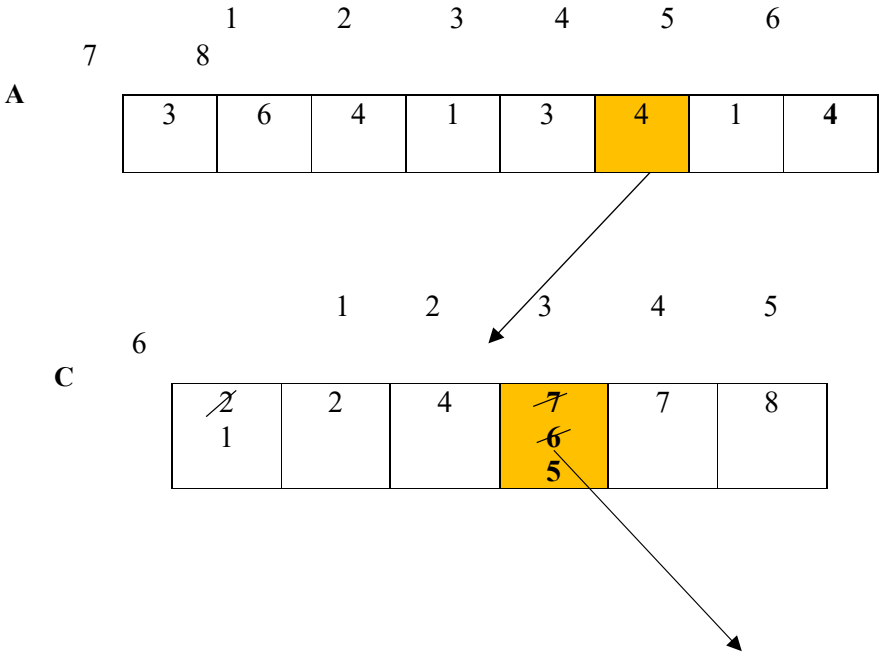


After performing this steps we will again go to array A and see the second last element of the array and follow the same steps

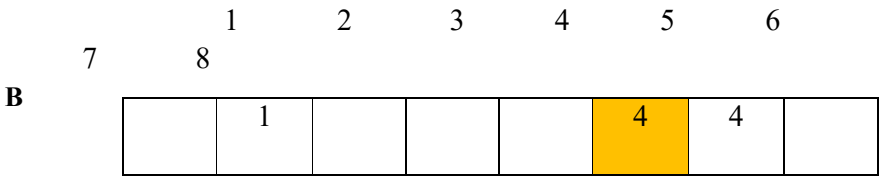
Space for learners:



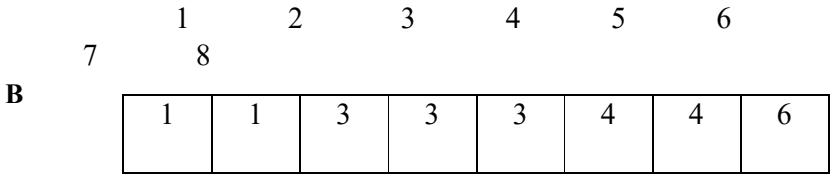
Here when we reach the third last element of array A i.e.A[6] we again get the element 4. But we need not to worry about its position in the output array as we have already placed the previous 4 in B[7] because now it will be placed in B[6] and this is the reason why we decrement the elements of C after placing the element in the correct position in array B.



Space for learners:



Finally repeating this steps we get the sorted array as:



6.3.1 Analysis of Counting Sort

The for loop of lines 1-2 takes time Θ(k), the lines 3-4 takes time Θ(n),the for loop of lines 6-7 takes time Θ(k) and the for loop of lines 9-11 takes time Θ(n). Thus the overall time is Θ(k+n). Counting sort is usually used when we have k=O(n) and in such cases the running time is Θ(n). As counting sort is a non-comparison sorting algorithm it is not affected by the arrangement of elements. The running time will remain the same even if the array is in reversed order, randomly sorted or already sorted.

STOP TO CONSIDER

- Counting sort is used to sort countable objects
- Used when linear time complexity is required
- Is applicable when a list of small integers with multiple counts are given.
- Useful when a finite and limited domain of variables are available

Space for learners:

6.4 RADIX SORT

Radix sort is a linear sorting algorithm used for integers. In radix sort the sorting of the elements are done by comparing digit by digit starting from the zero's place. The algorithm of the radix sort algorithm is given below:

RADIX-SORT(A,d)

1.for i ← 1 to d

2. do use a stable sort to sort array A on digit i.

Let us understand the logic of the algorithm with the help of an example. Let A be an array consisting of the elements [329,457,657,839,436,720,355]. First we will go through all the zero's place of the elements and sort them in ascending order.

A

0	32**9**
1	45**7**
2	65**7**
3	83**9**
4	43**6**
5	72**0**
6	35**5**

After going through the zero's place the array will be arranged in the following manner:

A

0	720
1	355
2	436
3	457
4	657
5	329
6	839

Now next we will check all the unit's place of the elements and arrange them accordingly

A

0	7**2**0
1	3**5**5
2	4**3**6

Space for learners:

3	4**5**7
4	6**5**7
5	3**2**9
6	8**3**9

A

0	720
1	329
2	436
3	839
4	355
5	457
6	657

Lastly we will go through all the hundred's place of all the elements and after arranging the elements we will get the final sorted array.

A

0	**7**20
1	**3**29
2	**4**36
3	**8**39
4	**3**55
5	**4**57
6	**6**57

A

0	329
1	355
2	436
3	457
4	657
5	720
6	839

6.4.1 Analysis of Radix Sort

Radix sort takes a list of n integers which are in base b (the radix) and so each number has at most d digits where $d=\lfloor(\log_b(k)+1)\rfloor d$ and k is the largest number in the list. For example, three digits are needed to represent decimal 104 (in base 10). It is important that

Space for learners:

radix sort can work with any base since the running time of the algorithm, O(d(n+b)), depends on the base it uses. The algorithm runs in linear time when b and n are of the same size magnitude, so knowing n, b can be manipulated to optimize the running time of the algorithm.

STOP TO CONSIDER

- Most suitable sorting algorithm in parallel machines

Check Your Progress

1. Between Counting and Radix sorting algorithm which algorithm would you prefer to be better and why?

6.5 MEDIAN AND ORDER STATISTICS

The k^{th}order statisticof a set of elements is the k^{th}smallest element. The minimum element of a set of elements is the first order statistic k=1 and the maximum element of a set of elements is the k^{th}order statistic k=n. A median of such a set of elements is a halfway point of the set. The median is unique when n is odd which is k=(n+1)/2. In case if n is even then we have two medians occurring at k = n/2 and k=n/2 +1. In any case median occurs at k= $\lfloor (n+1)/2 \rfloor$ which is the lower median and k = $\lceil (n+1)/2 \rceil$ is the upper median. Finally the problem is to select the i^{th}order statistic from a set of n distinct numbers. So the selection problem can be specified formally as follows:

Input: A set A of n (distinct) numbersand a number i, with $1 \leq i \leq n$.

Output: The element $x \in A$ that is larger than exactly i -1 other elements of A.

The selection problem can be solved in $O(n \lg n)$ time, since the numbers can be sorted using heapsort or merge sort and then simply index the i^{th} element in the output array.

Space for learners:

Let us see now how many comparisons are needed to find the maximum or the minimum element of an array. The way is to compare each element in pair to the other and then compare largest to the maximum element and lowest to the minimum element. Doing so, results in three comparisons per two element and finally the total running time will be O(3n/2).

Space for learners:

6.6 SELECTION PROBLEM USING RANDOMIZED SELECT ALGORITHM

The Selection Problem which is to find the i^{th} smallest element of a set can be solved using a practical randomized algorithm with O(n) expected running time. The algorithm is given below:

RANDOMIZED-SELECT (A,p,r,i) // return the i^{th} smallest element of A[p..r]

1) if (p == r)

2) then return A[p];

3) q := Randomized-Partition(A,p,r); // compute pivot

4) k := q-p+1; // number of elements <= pivot

5) if (i==k)

6) then return A[q]; // found ith smallest element

7) elseif (i < k)

8) then return Randomized-Select(A,p,q-1,i);

9) else Randomized-Select(A,q+1,r, i-k)

The Randomized-Select algorithm is similar to the quicksort algorithm as it creates partition in the array recursively. The only difference is that the quick sort algorithm recursively processes both sides of the partition whereas in Randomized Select algorithm it processes elements only from one side of the partition. The Randomized-Partition algorithm is similar to the partition process in Quick sort but here the pivot element is not the

rightmost element, but rather an element from A[p,r] that is chosen uniformly at random. The algorithm is given below:

Randomized-Partition(A,p,r)

1) i:= Random(p,r);

2) swap(A[i],A[r]);

3) Partition(A,p,r);

After RANDOMIZED-PARTITIONis executed in line 3 of the algorithm, the array A[p……r]is partitioned into two subarrays A[p…..q-1] and A[q+1….r] such that each element of A[p…q-1] is less than or equal to A[q] which in turn is less than each element of A[q+1…R]. Let us refer A[q] as the pivot element. Line 4 of RANDOMIZED-SELECT computes the number k of elements in the subarray A[p…q] that is the elements in the low side of the partition plus one for the pivot element. Line 5 then checks whether A[q] is the smallest element. If it is, then A[q] is returned. Otherwise, the algorithm determines in which of the two subarrays A[p….q-1] and A[q+1…r] the i^{th} smallest element lies. If i<k then the desired element lies on the low side of the partition, and it is recursively selected from the subarray in line 8. If i>k, however, then the desired element lies on the high side of the partition. Since k values that are smaller than the i^{th} smallest element of A[p…r] namely the elements of A[p…q] the desired element is the $(i-k)^{th}$ smallest element of A[q+1….r] which is found recursively in line 9. The code appears to allow recursive calls to subarrays with 0 elements.

STOP TO CONSIDER

- Randomized algorithm is widely used in cryptography

Space for learners:

6.6.1 Analysis of Randomized-Select Algorithm

If pivot q is k^{th} item in order, then algorithm is

If i = k, stop.

If i < k ⇒A[p..q − 1].

If i > k ⇒A[q + 1..r].

Let m = p − r + 1.
Note that if $k = p + \lfloor m/2 \rfloor$ was always true, this would halve theproblem size at every step and the running time would be at most

$n + n/2 + n/2 + n/2^2 + n/2^3 + \ldots\ldots = n(1 + 1/2 + 1/2^2 + 1/2^3 + \ldots\ldots) \leq 2n$

This isn't a realistic analysis because q is chosen randomly, so k isactually random number between p……r .

Again as we know

If i = k, stop.

If i < k ⇒A[p..q − 1].

If i > k ⇒A[q + 1..r].

Let m = p − r + 1.

Suppose that we could guarantee that $p + m/4 \leq k \leq p + 3/4\ m$.

This would be enough to force linearity because the recursive callwould always be to a sub problem of size ≤ 3/4 m and the runningtime of the entire algorithm would be at most

$n + 3n/4 + (3/4)^2 n + (3/4)^3 + \ldots\ldots\ldots \leq 4n$

Setting m = p − r + 1. We saw that if

$$p + m/4 \leq k \leq p + 3m/4$$

then the algorithm is linear.

While this is not always true, we can easily see that

$Pr\ (p + m/4 \leq k \leq p + 3m/4) \geq ½$

Space for learners:

This means that each stage of the algorithm has probability atleast 1/2 of reducing the problem size by 3/4.A careful analysis will show that this implies an O(n) expectedrunning time.

More formally, suppose t^{th} call to the algorithm is $A(p_t , r_t , i_t)$.

Let $M_t = r_t - p_t + 1$ be size of array in the sub-problem and k_t location of the random pivot in that sub-array. Note

- $p_1 = 1$, $r_1 = n$, $M_1 = n$
- $M_{t+1} \leq M_t - 1$
- Total cost of the algorithm is bounded by $\sum_t M_t$
- Set E_t to be event that is true if
  $p_t + M_t/4 \leq k_t \leq p_t + 3M_t/4$
  and false otherwise. Then
- $P_r(E_t) \geq ½$
- If $E_t$occurs then $M_{t+1} \leq 3M_t/4$

Since $M_1 = n$ and $M_{t+1} \leq M_t - 1$

- If $E_t \Rightarrow M_{t+1} \leq 3M_t/4$

Now let us define M'_tas follows-
$M'_1 = n$

If $E_t \Rightarrow M'_{t+1} = 3\ M'_t/4$. If (not Et)$\Rightarrow M'_{t+1} = M'_t$.
Then $\forall t, M_t \leq M'_t$.

In particular, since $\sum_t M_t$ bounds the algorithm's runtime,$\sum_t M't$ also bounds the algorithm's runtime.

Let us analyse another way for the running time of Randomized Select algorithm.

Let T (n) be the upper bound on the expected number of comparisons made by Randomized-Select(A, 1, n, i) for any i

$$T(n) \leq n + \sum_{k=1}^{n} \left(\frac{1}{n}.T(\max\{k-1, n-k\})\right)$$

$$T(n) \leq n + 2/n \sum_{k=\lfloor n/2 \rfloor}^{n=1} T(k)$$ which is a complicated recurrence

Space for learners:

We use the guess & induction method

Guess: T(n) ≤ cn for all n for some constant c

Induction step: Assume that T (m) ≤ c m for all m ≤ n − 1. Then try to show T (n) ≤ cn

T(n) ≤ n + 2/n $\sum_{k=\lfloor n/2 \rfloor}^{n=1} T(k)$

≤ n + 2/n$\sum_{k=\lfloor n/2 \rfloor}^{n=1} ck$

……..

≤ 3cn/4 +c/2 + n

We require 3cn/4 +c/2 + n≤ cn

Or n≥ 2c/(c-4)

If we choose c ≥ 12, then the induction step works for n ≥ 3.

Induction basis: T (1) ≤ c · 1, T (2) ≤ c · 2.

So if we choose c = max{12, T (1), T (2)/2}, then the entire proof works.

6.7 ADVERSARY ARGUMENT

An adversary is an opponent that akey-comparison algorithm plays against. Itsultimate goal is to maximize the number ofkey comparisons that the algorithm makeswhile constructing an input to the problem.At the beginning there is no restriction on theinput, but when the algorithm asks about apair (a, b) of keys the adversary must returneither a < b or a > b, then only the inputs thatare inconsistent with the answer will beremoved from further consideration. Now let us see how Adversarial Argument helps to bind the number of inputaccesses to an algorithm. Whenever an algorithm accesses a location in the input, the adversary argument

Space for learners:

fixes the value at that location. Whenever there is any future access by the algorithm to any of the fixed values is given for "free". The goal of the adversary is to fix values in such a way so that there is always two ways to fix the unfixed values so that the algorithm on the two resulting inputs has to output different answers. This simply implies that the algorithm cannot be terminated. If the adversary argument is able to continue this long enough, then there obviously a lower bound can be attained.

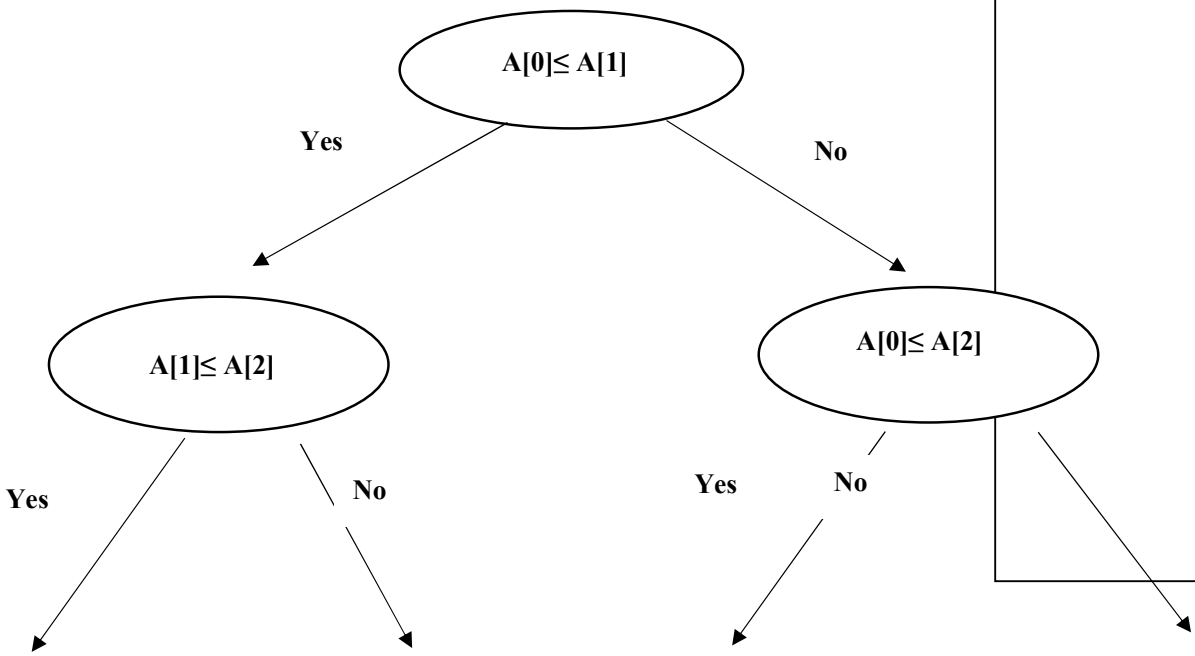
Space for learners:

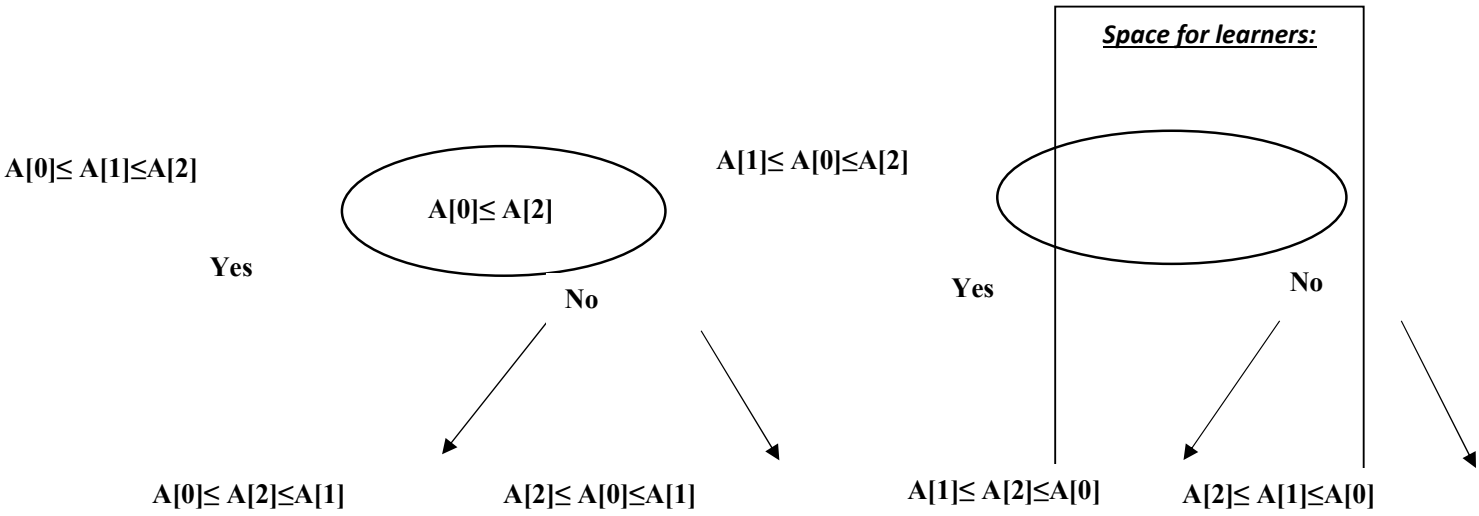
6.8 LOWER BOUND ON SORTING

In comparison sort algorithms given two elements A[i] and A[j] we perform the following test either A[i]>=A[j], A[i]<=A[j] and A[i] =A[j] to determine their order in the given array. The elements in the array are considered to be unique. One way of visualizing comparison sort is in terms of decision tree.

6.8.1 Decision Tree Model of Comparison Sorting

The comparisons made by a sorting algorithm can be represented by a decision tree which is a full binary tree. Let us go through an example in which we will perform insertion sort consisting of three elements using decision tree model. As there are 3 elements, a total of 3! = 6 is the possible permutation of the input elements, so the decision tree must consist of 6 leaves





Space for learners:

The internal nodes in the decision tree represent a comparison between any pair of elements A[i] and A[j] within the given range. If the condition given in the node is true, the left path is followed else the right path is followed. The execution of the algorithm is done by tracing the path from the root of the tree to a particular leaf. The left subtree conducts comparisons like A[i]< A[j] and the right subtree conducts comparisons whether A[i]>A[j]. The length of the longest path from the root of a decision tree represents the worst case number of comparisons that the corresponding sorting algorithm performs. The length is nothing but the height of the decision tree. A lower bound on the heights of decision trees is therefore a lower bound on the running time of any comparison sort algorithm.The following theorem establishes such a lower bound

Theorem: Any comparison sort algorithm requires Ω(nlgn) comparisons in the worst case.

Proof: Let us consider a decision tree of height hwith⌊ reachable leaves corresponding to a comparison sort on n elements. Because each of the n! permutations of the input appears as some leaf, we have n! ≤ ⌊.

Since a binary tree of height h has no more than 2^h leaves, we haven! ≤ ⌊ ≤ 2^h

which by taking algorithms, impliesh ≥ lg(n!)= Ω(nlgn)

Hence proved.

STOP TO CONSIDER

- Decision trees are used in classification and regression analysis

Check Your Progress

2. Define Median
3. What is a Selection Problem?
4. A decision tree is a full _____ tree

6.9 SUMMING UP

- Counting and Radix sorting algorithms are based on operations other than comparison.
- Counting sort is a sorting algorithm that finds maximum element of the given array and then creates an auxiliary array of the maximum element size
- The new auxiliary array which consists of the number of occurrences of the distinct elements in the array and the sorting is done by mapping the count as an index of the auxiliary array
- The time complexity of Counting sort algorithm is O(n).
- In radix sort the sorting of the elements are done by comparing digit by digit starting from the zero's place
- The time complexity of Radix sort algorithm isO(d(n+b))where n is the list of integers to be sorted, d is the number of digits, and b is the base of the number.
- A median of an set of elements is a halfway point of the set.

Space for learners:

- The median is k=(n+1)/2 if n is odd
- In case of even numbers there are two medians occurring at k = n/2 and k=n/2 +1.
- The selection problem can be specified aswhen a set A of n (distinct) numbers and a number i, with $1 \leq i \leq n$ is given the output will be an element $x \in A$ that is larger than exactly i -1 other elements of A.
- The selection problem can also be solved using a randomized algorithm whose expected running time is O(n).
- An adversary is an opponent that a key-comparison algorithm plays against.
- Its ultimate goal is to maximize the number of key comparisons that the algorithm makes while constructing an input to the problem.
- The comparisons made by a sorting algorithm can be represented by a decision tree which is a full binary tree
- Any comparison sort algorithm requires Ω(nlgn) comparisons in the worst case

Space for learners:

6.10 ANSWERS TO CHECK YOUR PROGRESS

1. Both counting sort and radix sort are non-comparison based sorting algorithms with a linear running time when the input elements are within a limited range. But counting sort is linear only when the range limit of data is linear whereas in radix sort even if the range of data is exponential, it works in linear time. This characteristic of radix sort is obviously an advantage over counting sort.
2. A median of an set of elements is a halfway point of the set.The median is k=(n+1)/2 if n is odd. In case of even

numbers there are two medians occurring at k = n/2 and k=n/2 +1.

3. The selection problem can be specified aswhen a set A of n (distinct) numbers and a number i, with $1 \leq i \leq n$ is given the output will be an element $x \in A$ that is larger than exactly i -1 other elements of A.
4. Binary

6.11 POSSIBLE QUESTIONS

1) Illustrate the operation of Counting Sort on the array A = <4,8,4,2,9,9,6,2,9>
2) How will you prove that Counting Sort is stable?
3) How many comparisons will be made to sort the array {2,7,1,3,8,2} using counting sort and why?
4) Is it possible to sort negative numbers using counting sort?
5) Illustrate the operation of Radix Sort on the array A = < 127,324,173,4,38,217,134>
6) Is Radix sort a stable sorting algorithm? Explain
7) Which sorting algorithm will you use to sort the integers between 123456 to 876543 in linear time and why?
8) With the help of an example explain the use of adversary arguments.
9) How will you solve the Selection Problem using Randomized algorithm? Explain
10) With the help of an example explain how decision tree can be used in comparison sorts?

Space for learners:

6.12 REFERENCES AND SUGGESTED READINGS

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.
- Rajendran(2017,Feb 10) *Medians and order statistics*, slideshare
- https://www.slideshare.net/rajendranjrf/medians-and-order-statistics-71991355
- Karlelgh Moore, *Radix Sort*, Brilliant,https://brilliant.org/wiki/radix-sort/
- Randomized Algorithms: Quick sort and Selection (2016,September 6) Retrieved from https://cse.hkust.edu.hk/mjg_lib/Classes/COMP3711H_Fall16/lectures/Randomized_ Handout.pdf
- Adversarial Lower Bounds, Retrieved fromhttp://www-student.cse.buffalo.edu/~atri/cse331/support/lower-bound/index.html

---×---

Space for learners:

BLOCK III:
PRIORITY QUEUE ADT, PARTITION ADT AND DATA STRUCTURE FOR EXTERNAL STORAGE OPERATIONS

UNIT 1: PRIORITY QUEUE ADT I

Unit Structure:

Space for learners:

1.1 INTRODUCTION

A tree is a nonlinear discrete data structure. A heap tree is one kind of binary tree. A heap tree may be a max heap or min-heap. This chapter introduces the heap tree and its properties. The extended priority queue is also discussed in this unit for example. The algorithm of max and min heap insertion and deletion is discussed in this unit. The algorithm complexities of the min and max heap along with its application are reported in this article.

1.2 UNIT OBJECTIVES

After going through this unit, you will be able to know—

i) About heap trees and their properties.
ii) About the extended priority queue.
iii) About min and max heap.
iv) About the algorithm of min and max heap tree.

1.3 HEAP

Heap is a tree where nodes are in a specific order. If node a is the parent node of b, then the value of node a will be either greater or

smaller than b. A specific order is maintained to form the tree. The root node of the heap tree is always compared with the children node. Based on the comparison, a heap tree is divided into two types.

i) Max Heap Tree
ii) Min Heap Tree

In the max heap tree, the root node is maximum in the tree whereas the root node is minimum in the min-heap tree. The number of children of a node in the heap tree depends on the type of the heap tree. If a heap node contains two children, then the heap tree is a binary heap tree. A binary heap has the following properties.

i) It is a complete binary tree. It means that except for the last level, other levels are filled.
ii) A binary heap may be either min or a max heap.

As mentioned above, the binary heap is a complete binary tree, so it is represented by an array. The root element will be at a[0].Below the indexes of other nodes are shown for the i^{th} node, i.e., a[i]:

i) a[(i-1)/2] - returns the parent node

ii) a[(2*i) +1] - returns the left child node

iii) a[(2*i) +2] - Returns the right child node

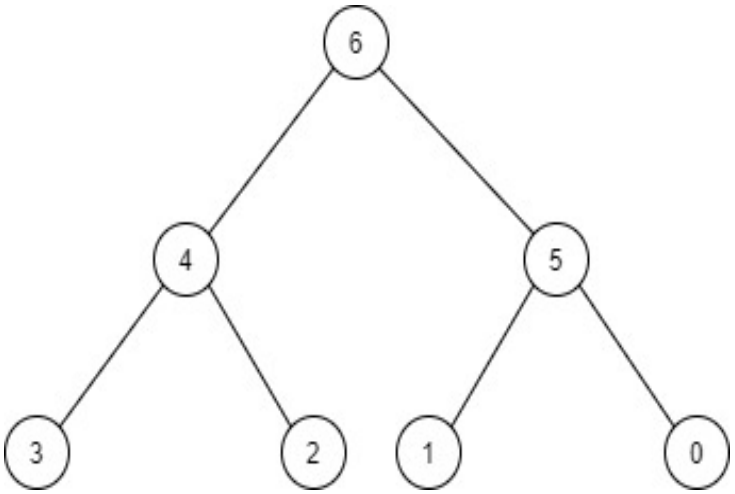


Figure 1.1: Max heap tree

So, if you perform the traversing based on the above array representation, then Figure 1.1 will represent as a follows

.

Space for learners:

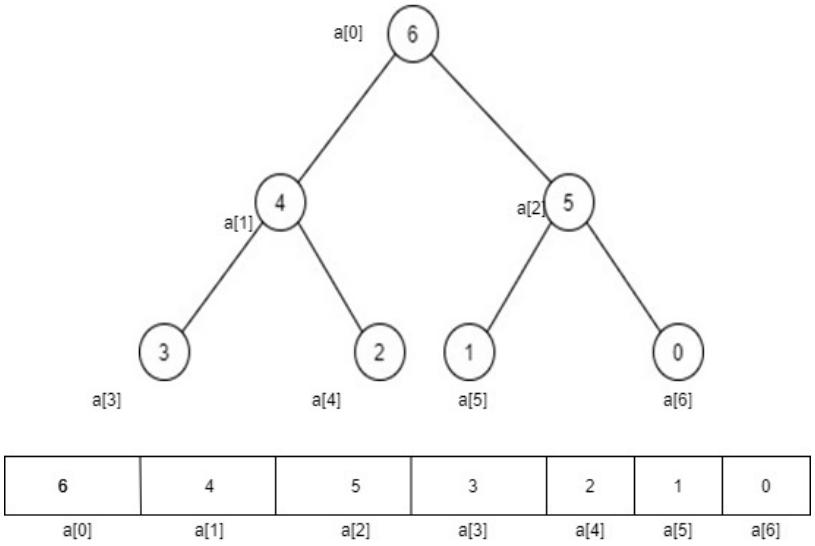


Figure 1.2: Array Representation of Heap

Figure 1.1 shows a binary max-heap tree where the root node 6 is greater than all the other nodes. In the 1st level, two nodes 4 and 5 are more than the leaf node 3, 2, 1, and 0. Figure 1.2 shows the binary min-heap where the root is the minimum one as compared to the other nodes. The root is 2, the children of the root are 4, and 6 which are more are more than root. The leaf nodes are 8 and 10 which are more than the whole node. So, this is a min-heap.

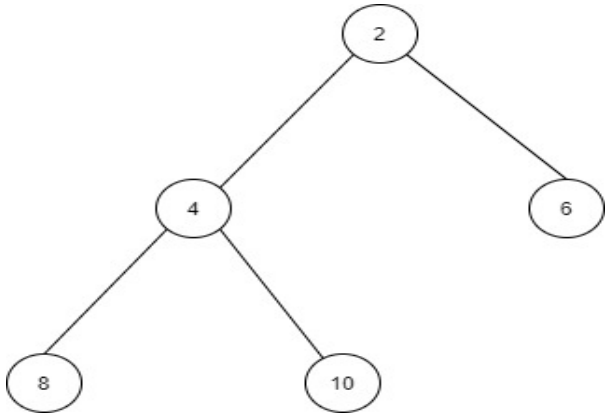


Figure 1.3: Min Heap Tree

Space for learners:

1.4 HEAP TREE-BASED EXTENDED PRIORITY QUEUE

Space for learners:

An extended priority queue is an extension of a queue. The priority queue has the following properties.

i) Every item in the queue has a priority associated with it.
ii) The element with the highest priority is deleted (dequeued) first.
iii) In case of the same priority, the items are served according to their order.
iv) The element with the highest priority is placed in the front queue.

A priority queue can be implemented using an array and link list. But array and link implementation of priority queue take O (1) times for array and link list insertion of the priority queue. But, finding the highest priority of and link list insertion of priority queue takesO(n) time. In heap tree implementation, the complexity is O(1) and O(logn). So, it is less than the time taken by the array and link list implementation of the priority queue.

Based on the heap structure, the priority queue is of two types.

i) Max Priority Queue
ii) Min Priority Queue

Let's take an example of an array of size 5 i.e., a= {3,6,1,5}. Let's insert all the elements in the max priority queue.

i) First Insert 3 in the queue. As the queue is initially empty, so no issue to insert 3.
ii) Now for 6, as 6 is greater than 4, so it will come in front of the queue.
iii) In the next step, 1 will be inserted in the back of the max priority queue as its value is less than 3 and 6.
iv) Now, 5 will be inserted in between 3 and 6. It will be inserted after 6.

6	5	3	1

Like if you consider the above example for the min priority queue. Then the first element will be 1, thenthe second element will be 3.

After 3, the next element will be 5, and finally, the last element will be 6.

The max priority element has the following operations based on the heap tree.

i) Maximum(a): It returns maximum element from the array (a).

```
int Maximum (int a[ ])
{
return a[1];
}

// The complexity of the Maximum function is O(1).
```

ii) Extract_Maximum (a): It removes and returns the maximum element from the array (a).

```
intExtract_Maximum (int a[ ])
{
        if(length == 0)
{
        cout<< "queue is empty";
        return -1;
}
        int max = a[1];
        a[1] = a[length];
        length = length -1;
        max_heapify(a, 1);
        return max;
}

// The complexity of the Extract_Maximum Function is O(logn)
```

iii) Increase_Val(a, i, val) - It increases the key of element stored at index i in array (a) to new value val.

```
voidIncrease_Value (int a[ ], int i, intval)
{
if(val<a[ i ])
{
```

Space for learners:

```
cout<< "The inserted element is less than other value, so cannot
insert";
return;
}
a[i ] = val;
while( i> 1 and a[ i/2 ] <a[ i ])
{
swap|(a[ i/2 ], a[ i ]);
i = i/2;
}
}
// The complexity of this algorithm is O(logn).
```

iv) insert_val (a, Val) - It inserts the element with value val in array(a).

```
voidinsert_value (int a[ ], intval)
{
length = length + 1;
a[ length ] = -1;
Increase_Val (a, length, val);
}

// The complexity of the insert_value algorithm is O(logn)
```

Check Your Progress-I

1. What is a heap tree?
2. What is a binary heap tree?
3. True or False
   i) a[(2*i) +1] - returns the left child node
   ii) Binary Heap tree contains two child nodes
   iii) The complexity of heap tree insertion is O(n).
   iv) The complexity of extract maximum algorithm of heap tree is O(logn).

1.5 MIN HEAP

In the min-heap tree, the root element must be less than the other elements. It uses the ascending priority. In min-heap, the smallest

Space for learners:

element has the priority and it is popped out first. The following is an example of a min-heap tree.

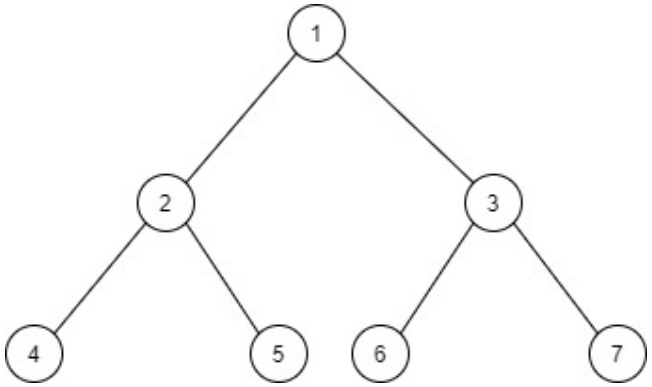


Figure 1.4: Min Heap Tree

In the Figure 1.4, the elements {1, 2, 3, 4, 5, 6, 7} are inserted as follows.

i) The first element is 1, as the heap is null, you can directly insert 1.

ii) The second element is 2 and it is placed in the second position of the tree as per array position.

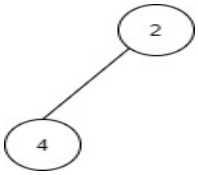
iii) The third element is 3, and it is more than 1 and 2. So no need to perform happifies.

iv) Then you can insert 4, 5, 6, and 7 by following the steps i) to iii).

Let's take another example. The array is a = {2, 4, 1, 6, 3}

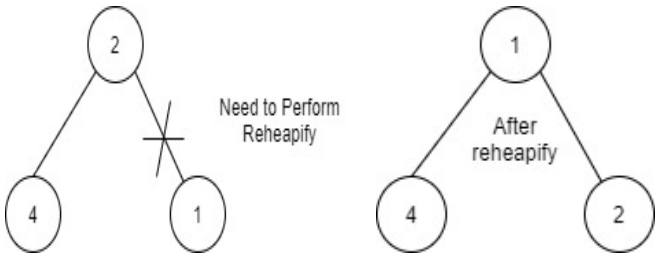
i) At first, insert the first element in the min-heap.

ii) Now, you can insert 4. As 4 is more than 2, 4 will be a child of 2 and it will be placed left of 2.

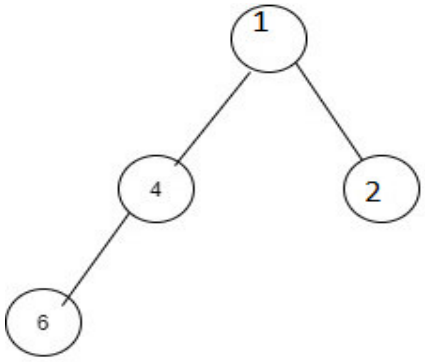


iii) The next element is 1. If you insert 1 on the right of 2, it will violate the property of min-heap. So, you need to heapify it. After heapify, the 1 will go to place 2 and 2 will come to the place of 1.

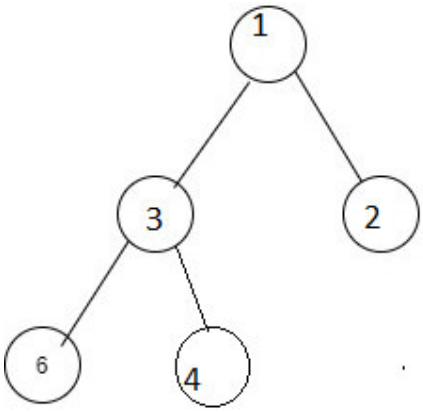
Space for learners:



iv) The next element is 6, it will go to the left of 4.



v) The next element is 3. If you add 3 as right 4, it will violate the property of min-heap. You need to perform the heapify. After heapify, element 3 will take 4 positions and 4 will come as a right of 3.



Now, we will discuss the complexity of the insertion operation of the min-heap in the following steps.

i) If you add a node is at a level of height h, then the complexity of adding is O(1).

Space for learners:

ii) The complexity of heapifyis O(h)
So, total complexity: O (1) + O(h) = O(h)

iii) For a complete binary tree, its height h = O (log n), where n is the total no. of nodes.

iv) Therefore, the overall complexity of the insert operation is O (log n).

After insertion, follow the following steps for deletion.

i) Delete the root node

ii) Then, the last node according to the position of the tree will be placed as a root node.

iii) The last node may not follow the property of the min-heap. So, heapify again.

Let's consider the previous example of the tree.

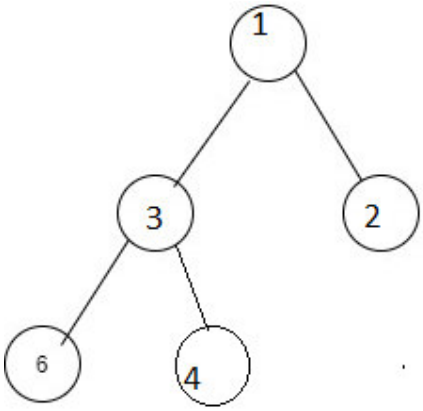


Figure 1.5: Min Heap Tree

Now, the deletions of nodes are done as follows.

i) Delete node 1 from min-heap and place node 4 in the root position.

ii) The new root 4 is not following the property of the min-heap. So, you need to perform heapify.

iii) After heapify, 2 will be placed as root and 4 will take the position of 2. In this step, all the nodes will follow the min-heap property.

iv) Now, delete node, 2 and place node 4 as the root node.

v) Now, delete node 4 and place 6 as root.

vi) Finally, delete node 6.

The complexity of the deletion operation can be explained as follows.

Space for learners:

i) If you delete a node from a heap with height h, then the Complexity of swapping parent node and leaf node is O(1)

ii) The complexity of heapifying is O(h). So, the complexity: O(1) + O(h) = O(h)

iii) For a complete binary tree, its height h is O (log n), where n is the total no. of nodes.

iv) Therefore, the overall complexity of the delete operation of the min-heap is O (log n).

Space for learners:

1.6 MAX HEAP

The max is the reverse of the min-heap. In the max heap tree, the root element must be greater than the other elements. It uses the descending priority. In a max heap, the greatest element has the priority and it is popped out first. The following is the example of the max heap tree.

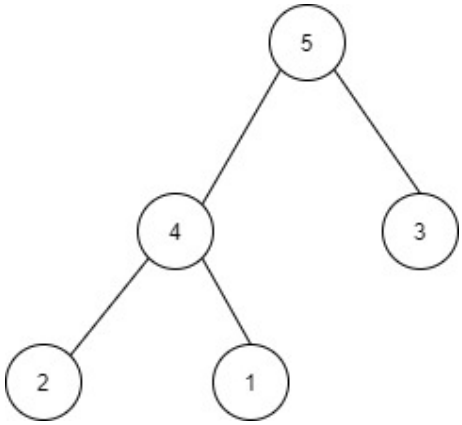


Figure 1.6: Max Heap Tree

In Figure 1.6, the root node of the tree is 5 that is the greatest value of the tree. Here, the list of arrays is {5.4.3.2, 1}. So, the insertions of the nodes take place as follows.

i) Insert node 5 in the tree. As the tree is empty, so you can directly insert node 5 in the tree.

ii) The 2nd node is 4. It will go to the left of 5.

iii) The next node is 3. So, 3 will go to the right of 5 by maintaining the max heap tree property.

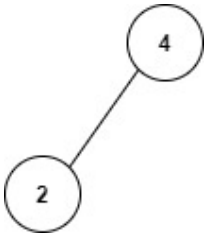
iv) Node 2 will go to the left of 4 by maintaining the max heap tree.

v) The last node 1 will be placed on the right of 4.

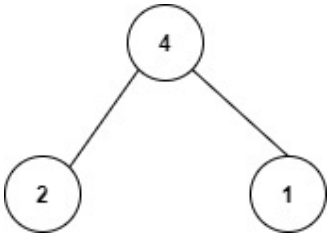
Here, you can all the nodes maintain the max heap tree property. During the time of node insertion, not a single node violates the property of the max heap tree. But it will not happen in all cases. To better understand, let's take an example.The array is now a = {2, 4, 1, 6, 3}. For the max heap tree formation, the following steps are implemented.

i) First, insert the 2 in the empty tree.

ii) In this step, element 4 is added to the tree. And it will go to the left of 2. It has violated the property of the max heap. So, you need to perform the heapify. After heapify, node 4 is in the root position and 2 is placed as a left of 4.

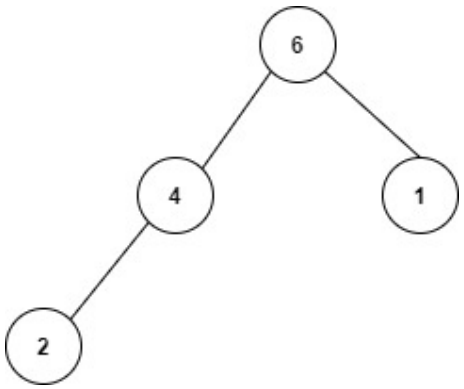


iii) The next element 1 is placed as a right of 4. So, no issue and no need to perform the heapify.

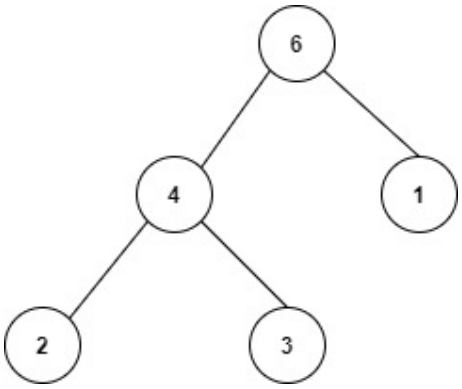


iv) The next element is 6. It will place as a left of 2. But it violates the rule of max heap. So, heapify is required. After heapify, the 6 and 2 are interchanged to their position. But 6 can be left of 4 because node 6 is more than 4. Again, you need to perform the heapify. After heapify, the node 4 and 6 will be interchanged. Finally, the tree looks like below.

Space for learners:



v) The final element is 3. It is placed as a right of 4. It is not violating the property of the heap tree. So, the final tree is as follows.



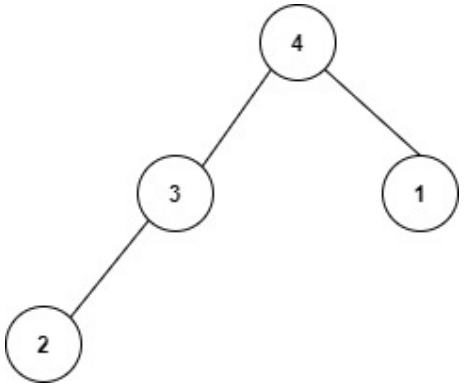
After insertion, the following steps are used for the deletion in the max heap tree.

i) Delete the root node (greatest priority).
ii) Then, the last node according to the position of the tree will be placed as a root node.
iii) After interchanging, its max heap tree property is not maintained, perform the heapify.
iv) Follow the above steps.

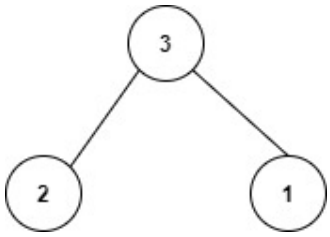
So, consider the above three and perform the deletion operation in the tree.

i) The first deleted node is 6. After deletion, 3 will be placed in the position of 6. But it violates the rule of max heap. So,you need to perform the heapify. After heapify, 4 will act as root, and 3 will be as left of 4.

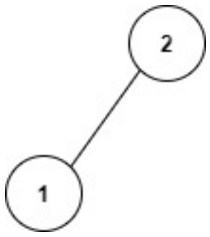
Space for learners:



ii) Now, delete node 4 from the tree, and 2 will take the position of 4. Due to max heap property violation, you need to interchange the value of 2 and 3. So, 3 is the root node and 2 will be on the left of 3.



iii) Now, delete node 3 from the tree. Node 1 will take the position of 3. You need to perform the heapify. After heapify, the root will be 2, and 1 will be the left of 2.



iv) Now, delete 2, and node 1 will be a single node in the tree. Finally,deletenode 1.

The insertion and deletion of the max heap take the same time as like min-heap. The complexities of the insertion operations of the max heaparein the following steps.

i) If you add a node is at a level of height h, then the complexity of adding is O (1).

ii) The complexity of reheapifyis O(h)
So, total complexity: O (1) + O(h) = O(h)

Space for learners:

iii) For a complete binary tree, its height h = O (log n), where n is the total no. of nodes.

iv) Therefore, the overall complexity of the insert operation of the max heap is O (log n).

The complexity of the deletion operation of the max heap can be explained as follows.

i) If you delete a node from a heap with height h, then the Complexity of swapping parent node and leaf node is O (1)

ii) The complexity of heapifying is O (h). So, the complexity: O (1) + O (h) = O (h)

iii) For a complete binary tree, its height h is O (log n), where n is the total no. of nodes.

iv) Therefore, the overall complexity of the delete operation of the max heap is O (log n).

Check Your Progress-II

4. What is a max heap ?
5. What is a min heap ?
6. What is the deletion complexity of min and max heap?
7. True or False
   i) Descending priority queue can be implemented using Max heap
   ii) Ascending priority queue can be implemented using min heap.
   iii) Min heap can be used to implement selection sort.
   iv) The ascending heap property is A[Parent(i)] <= A[i]
   v) The procedure FindMin() to find the minimum element and the procedure DeleteMin() to delete the minimum element in min heap take logarithmic time.

Space for learners:

1.7 SUMMING UP

Space for learners:

i) Heap is a tree where nodes are in a specific order. If node a is the parent node of b, then the value of node a will be either greater or smaller than b.

ii) Based on the comparison, a heap tree is divided into two types.
   a. Max Heap Tree
   b. Min Heap Tree

iii) A binary heap has the following properties.
   a. It is a complete binary tree. It means that except for the last level, other levels are filled.
   b. A binary heap may be either min or a max heap.

iv) An extended priority queue is an extension of a queue. The priority queue has the following properties.
   a. Every item in the queue has a priority associated with it.
   b. The element with the highest priority is deleted (dequeued) first.
   c. In case of the same priority, the items are served according to their order.
   d. The element with the highest priority is placed in the front queue.

v) Based on the heap structure, the priority queue is of two types.
   a. Max Priority Queue
   b. Min Priority Queue

vi) In the min-heap tree, the root element must be less than the other elements. It uses the ascending priority. In min-heap, the smallest element has the priority and it is popped out first.

vii) The overall complexity of the min-heap insert operation is O (log n).

viii) Therefore, the overall complexity of the delete operation of the min-heap is O (log n).

ix) The max is the reverse of the min-heap. In the max heap tree, the root element must be greater than the other elements. It uses the descending priority. In a max heap, the greatest element has the priority and it is popped out first.

x) The overall complexity of the max heap insert operation is O (log n).

xi) Therefore, the overall complexity of the delete operation of the max heap is O (log n).

Space for learners:

1.8 ANSWER TO CHECK YOUR PROGRESS

1. Heap is a tree where nodes are in a specific order. If node ais the parent node of b, then the value of node a will be either greater or smaller than b. A specific order is maintained to form the tree.
2. If a heap node contains two children, then the heap tree is a binary heap tree.
3. i) True ii) Trueiii) Falseiv) True
4. In the min-heap tree, the root element must be less than the other elements. It uses the ascending priority. In min-heap, the smallest element has the priority and it is popped out first.
5. The max is the reverse of the min-heap. In the max heap tree, the root element must be greater than the other elements. It uses the descending priority. In a max heap, the greatest element has the priority and it is popped out first.
6. O(logn)
7. i) True ii) True iii) True iv) True v) true

1.9 POSSIBLE QUESTIONS

Short answer type questions:

i) What is a heap tree?

ii) What is a binary heap tree?

iii) What is a max heap tree?

iv) What is a min-heap tree?

v) What is the insertion complexity of the min and max heap tree?

vi) What is a priority queue?

vii) How does heap use for priority queue?

viii) What is happify?

ix) What is ascending priority?

x) What is descending priority?

xi) What is max and min priority queue?

Long answer type questions:

i) Construct the min-heap for the following elements.
   a) 1,5,2,3,8,15,41,14
   b) 2,5,41,24,23,7,8,9,12
   c) 1,2,3,4,5,6

ii) Construct the max heap for the following elements.
   d) 1,5,2,3,8,15,41,14
   e) 2,5,41,24,23,7,8,9,12
   f) 1,2,3,4,5,6

iii) Write the algorithms for the different operations of max heap.

Space for learners:

1.10 REFERENCS AND SUGGESTED READINGS

- Karumanchi, Narasimha. *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*. almohreraladbi, 2011.
- Thareja, Reema. Data structures using C. Oxford University Press, Inc., 2011.
- Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2022.

---×---

UNIT 2: PRIORITY QUEUE ADT- II

Unit Structure:

Space for learners:

2.1 INTRODUCTION

A heap tree is a binary tree where the root node is either maximum or minimum. Here, the priority queue is discussed concerning the binomial heap. A binomial heap is a priority queue where the heap is merged. Along with the binomial heap, the Fibonacci heap is also discussed. It is also used for priority queues and it has good amortized analysis than other priority queues such as binomial heap.

The amortized analysis of the priority concerning binomial and Fibonacci heap is also discussed in this unit.

Space for learners:

2.2 UNIT OBJECTIVES

After going through this unit, you will be able to know

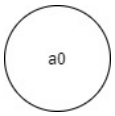
i) About binomial heap and their properties.
ii) About the Fibonacci heap and its properties.
iii) About amortized analysis of the binomial and Fibonacci heap.

2.3 BINOMIAL HEAP

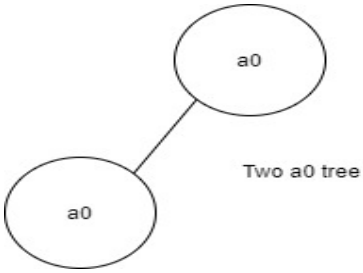
A binomial heap is a priority queue data structure thatallows pairs of heaps to be merged. It is important because of the mergeable heap abstract data type. The binary heap is used for priority queueimplementation. It is an extension of binary heap that provides merge operation of the heapalong with the other operations ofbinary Heap.

Before discussing the binomial heap, let's discuss the binomial tree. A binomial tree of order 0 has 1 node. A binomial tree of order 1 has 2 nodes whereas the 4 nodes are present if the order of the tree is 2. It has 2^k nodes in the tree. The depth of the tree is K where ${}^kC_i$ nodes are present at a depth i. Suppose the root of the binomial tree is '*ak'* and its children have a degree of k-1, k-2, ….., etc.Let's understand the binomial tree through an example.

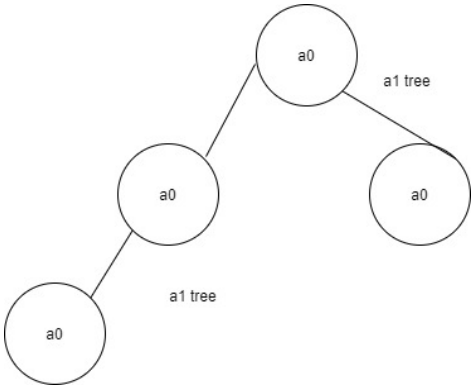
i) If a0, where k(order) is 0. It means that only one node is present in the tree as shown below.



ii) Let's the next element is a1. It means that k =1. So the value of k-1 is equal to 0. Two binomial trees of a0 are here in which one a0 becomes the left subtree of another a0.

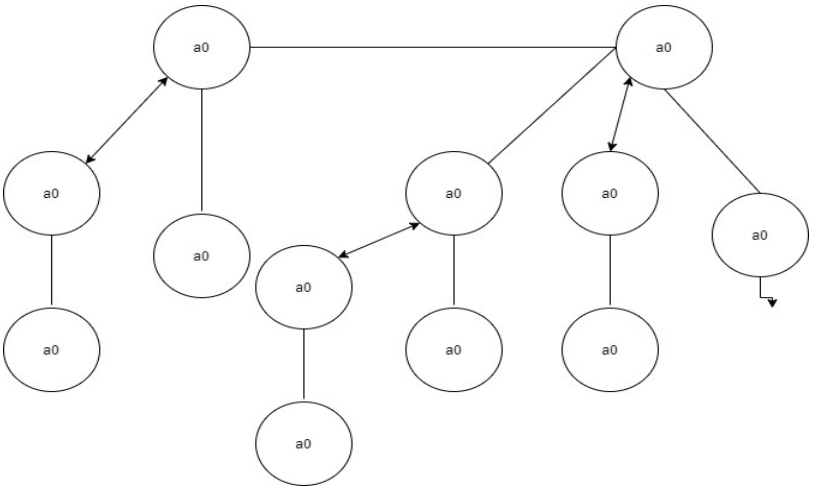


iii) Now for a2, if K=2, it means that k-1 is equal to 1. Two binomial trees of a1 are here in which one a1 becomes the left subtree of another a1.



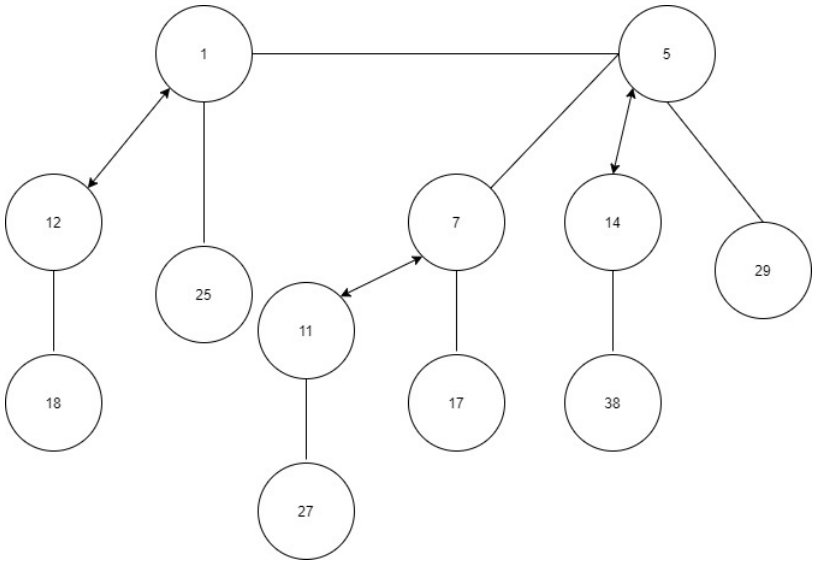
A binomial heap is a collection of binomial trees where each binomial tree is a min-heap tree. It has the following properties.

i) Each binomial heap obeysthe properties of the min-heap.
ii) For K (non-negative) value, there should be atleast one binomial tree where the root has degree k.



Space for learners:

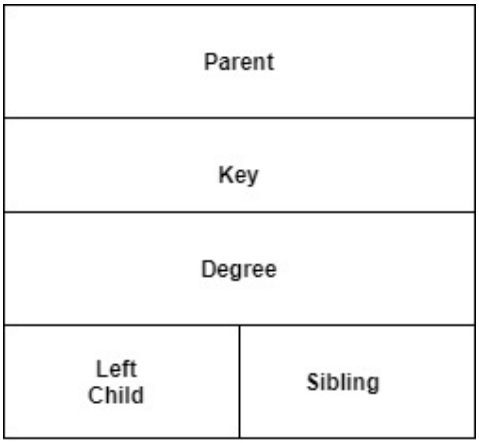
In the above example, two binomial trees are merged. So, according to the properties of the binomial heap, the tree should follow the properties of the min-heap tree and the degree root of the tree should be K. Here, the degree is 3. If you insert some values in the tree and it now the tree is a min-heap binomial tree.



Space for learners:

2.4 BINOMIAL HEAP IN MEMORY

The binomial heap is represented as follows in memory. The first block of the memory represents the parent or root of the tree. The second block store the key values whereas the third block store the degree of the tree. The fourth block has two parts. The first part store the value of the left child and the right part store the value of the siblings.



Stop to Consider

If you want to create a binomial heap of 'n' nodes, then that can be easily defined by the binary number of the n. For example: if youwant to create the binomial heap of 5 nodes. Then initially you have to find the binary number of the 5, and i.e. 101. From the binary number, you can observe that 1 is available at the 0 and 2, positions. Therefore, the binomial heap with 5 nodes will have a0 and a2binomial trees.

Space for learners:

Check Your Progress-I

1. What is a binomial tree?
2. What is a binomial heap ?
3. True or False
   i) Binomial Heap is min heap
   ii) To create the binomial heap of 9 nodes, two binomial tree is required.
4. How many binomial tree will be required to construct a binomial heap of node 13?

2.5 COMPLEXITIES OF BINOMIAL HEAP OPERATIONS

The binomial heap has the following operations.

i) Creating a binomial heaptakesO(1) time because creating a heap will create the head of the heap without any element.

ii) Finding the minimum key from a binomial heap means extracting the minimum value. Here you need to compare only the root node of all the binomial trees to find the minimum key with a complexity value of O(logn).

iii) Union of two binomial heaps means finding the union of two binomial heaps. The time complexity for finding a union is O(logn).

iv) Inserting a node in a binomial heap takes O(logn).

v) Extracting the minimum key from the binomial heaptakesO(logn).

vi) Deleting a node from the binomial heap takesO(logn).

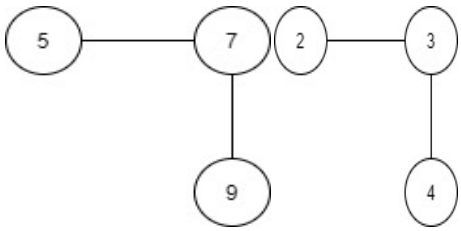
Space for learners:

2.6 UNION OF TWO BINOMIAL HEAP

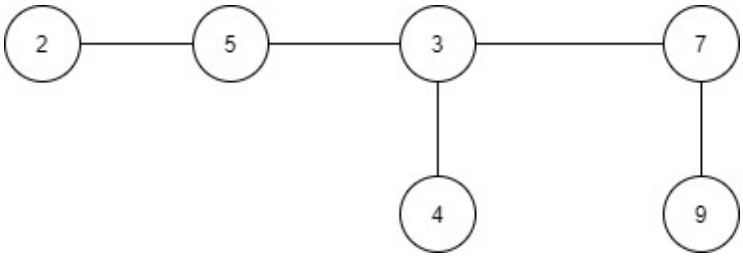
Two binomial heapcan be united using the following case.

1. If degree[a] is not equal to degree[next a] then move the pointer ahead.
2. If degree[a] = degree[next a] = degree[sibling(next a)] thenrearrarange.
3. If degree[a] = degree[next a] != degree[sibling[next a]]and key[a] < key[next a] then remove [next a] from root and attached to a.
4. If degree[a] = degree[next a] != degree[sibling[next a]]and key[a] > key[next a] then remove a from root and attached to [next a].

Let's understand the union by considering the following example

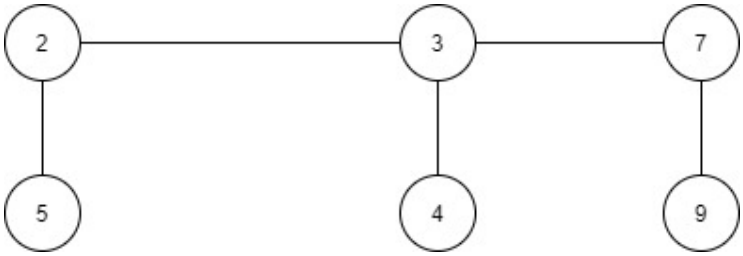


In the above example, you have two binomial heaps. For union, you have to find a minimum of two binomial heaps. So the minimum is 2. By finding the minimum values, the binomial heap will be presented in a sorted manner according to its order as presented below.

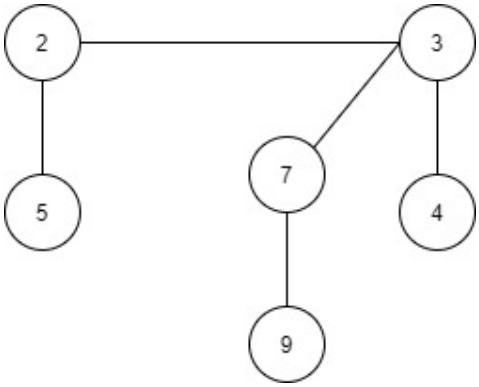


Now, according to case II, the degree[2] = degree[3] are the same, so, you need to rearrange. It means these two nodes will be added.

After addition, the 2 will act as root, and 5 will come as the child of 2 according to the min-heap rule.



Now you have 3 binomial heaps with the same degree. To apply the same rule in the last two binomial heaps. It means that 7 will be to the left of 3.

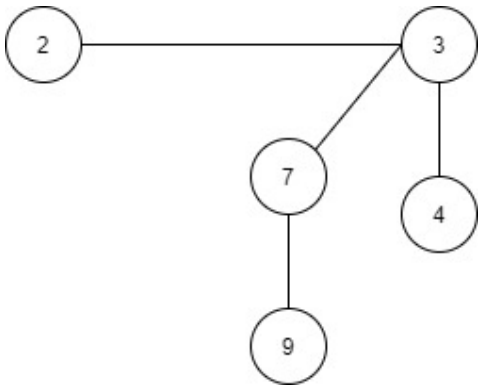


After that, the 4 cases will not fall in the above tree. So this is the union of the binomial heap.

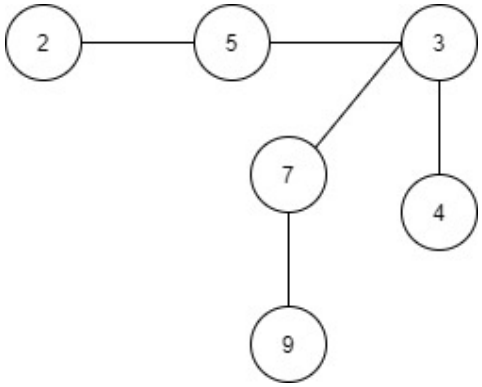
2.7 INSERT A NEW NODE IN BINOMIAL HEAP

The insertion of a new node in the binomial heap takes O(logn) times. Let's understand the insertion by considering the following examples.

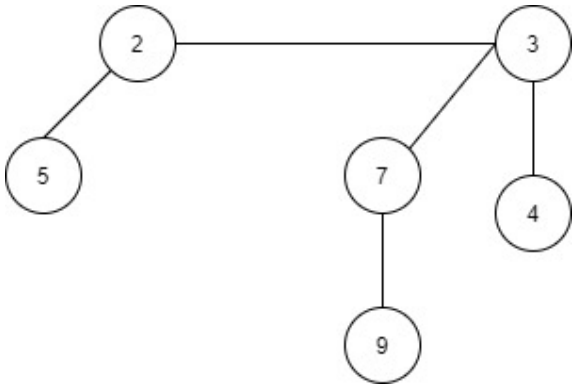
Space for learners:



In the above binomial heap, node 2 is attached to the head of the heap. Lets you want to insert node 5 in a binomial heap. In the insertion, you should following the rule of min-heap. After insert 5, the 5 will be removed from the head of the heap and place after 2 because 2 is the minimum one among all in the same order.

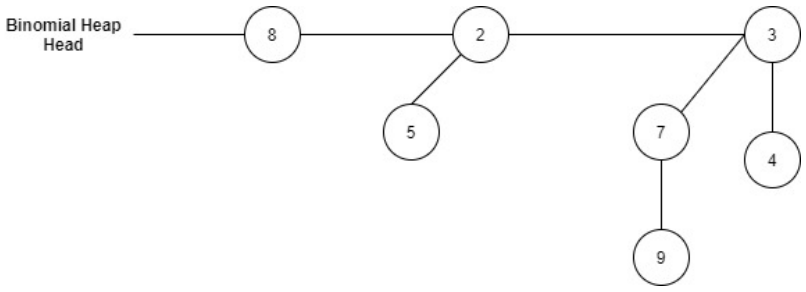


Now, you have inserted 5 in the binomial heap. As the degree of 2 and 3 are the same. So you need to check the possibilities of the above 4 cases. Case II is possible. So after applying case II in the binomial heap, 5 will come as a left child of 2. Now the degree is not the same. So this is the final tree.



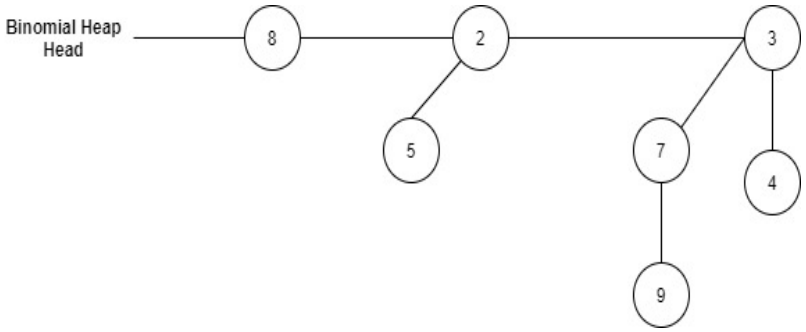
Space for learners:

In this above tree if you want to add a node 8. You can directly add 8 in the tree because 8 is the minimum one in the degree A0 or B0. Apart from 8, no such node is present in the tree having the same degree.



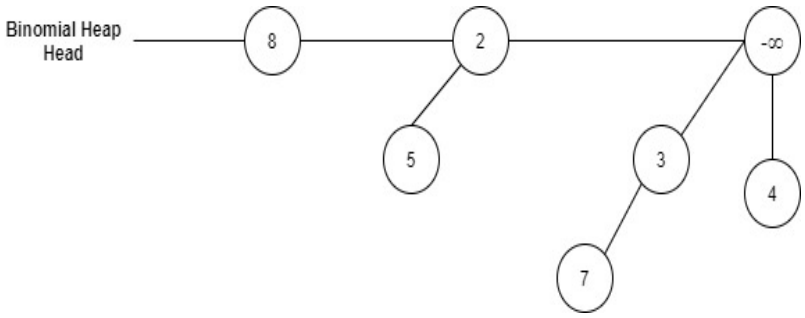
2.8 DELETE A NEW NODE IN BINOMIAL HEAP

During the deletion, you need to apply two operations of the binomial heap, i.e, extract min and decrease. Let's understand the deleting a node from a heap.

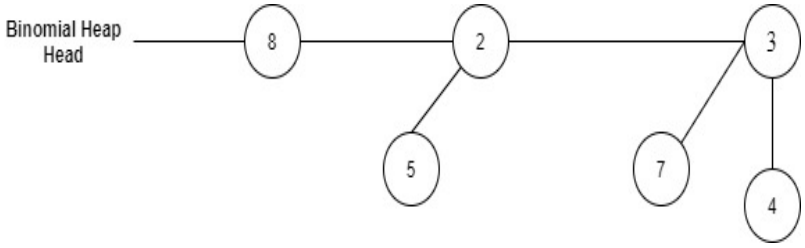


Let the deleting node is 9. Here the node 9 will be replaced by the smallest value. The smallest is -∞. So, delete node 9 and place -∞ in place of 9. As -∞ is the smallest one it will go up and take the position of 3 as given below according to the rule of min-heap.

Space for learners:



Now, use the extract min algorithm to extract the -∞ and finally, the binomial heap will be as follows.



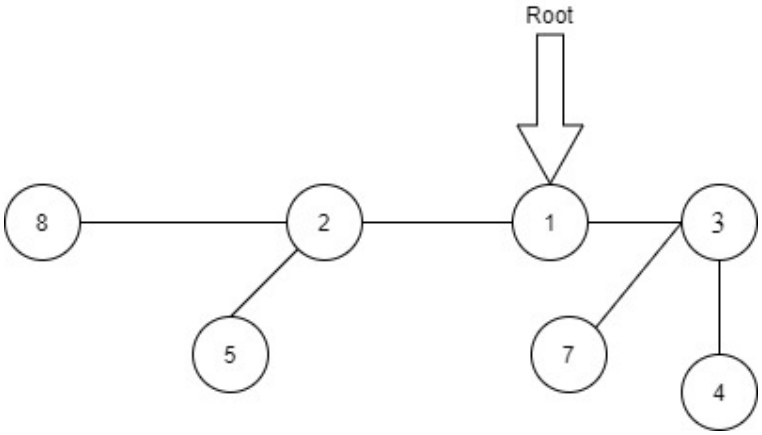
2.9 FIBONACCI HEAP

In the above section, you learn that heaps are used for implementing priority queues. The binomial heap takes more computational time in its all operation. To reduce the computational time, the Fibonacci heap is used. The Fibonacci heap takes the following amortized time for the different operations.

i) Finding min will take Θ(1)
ii) Delete min will take O(logn)
iii) Insert element in the Fibonacci heap will take Θ(1)
iv) Decrease Key will take Θ(1)
v) Merging will take Θ(1)

Binomial heap takes O(logn) time, in all the cases except the min finding. As compared to the binomial heap, the Fibonacci heap is more efficient. Like binomial heap, the Fibonacci heap is a collection of trees with within or max-heap property. A Fibonacci heap can be any order. The orders are mixed and the root is pointed out as shown below.

Space for learners:



The above tree is a Fibonacci heap. The tree is min-heap, but the orders of the nodes are mixed up. Though the order of 8 and 1 are the same they are not present together like a binomial heap. The minimum node is pointed out and that is the root node. In the heap, node 1 is the minimum one

2.10 INSERTING NEW NODE IN FIBONACCI HEAP

Inserting a new node at the Fibonacci heap is an easier method than a binomial heap. The insertion should take place according to the following order.

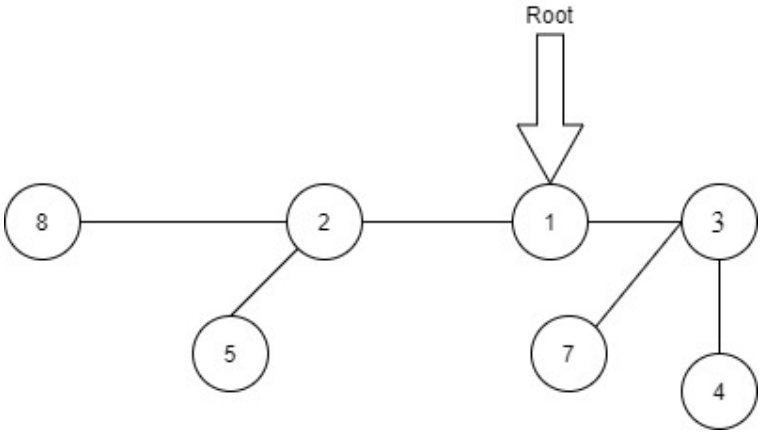
i) Create a new node ‘n’.

ii) If the heap is empty, ‘n’ be the root node in the root list.

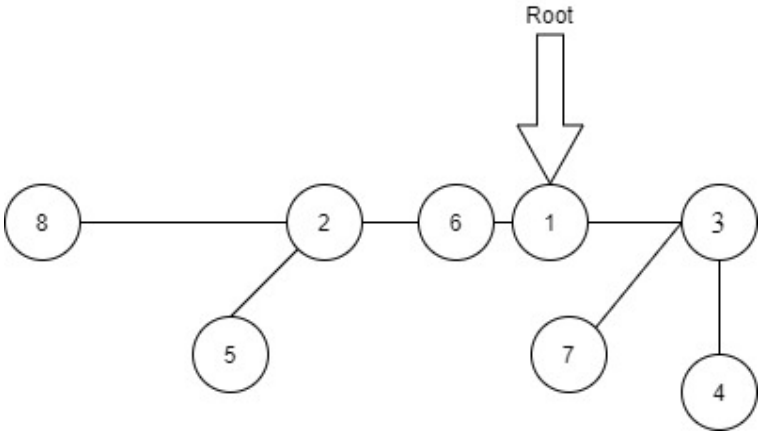
iii) Otherwise, insert ‘n’ into the root list and update the heap.

Space for learners:

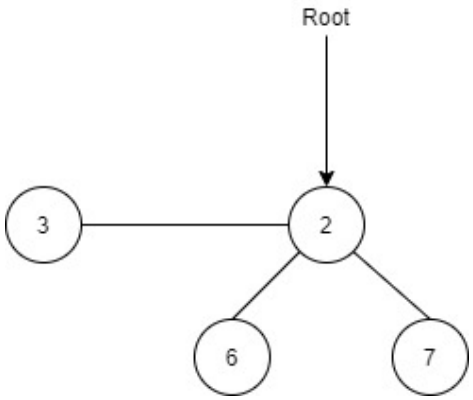
Let's consider the following tree.



Let's, you want to insert 6 in the heap. As the heap is already present and you can consider node 6 as a singleton tree and consider rule no 3 for the case. After following rule 3, the node is 6 will be added as left of 1.



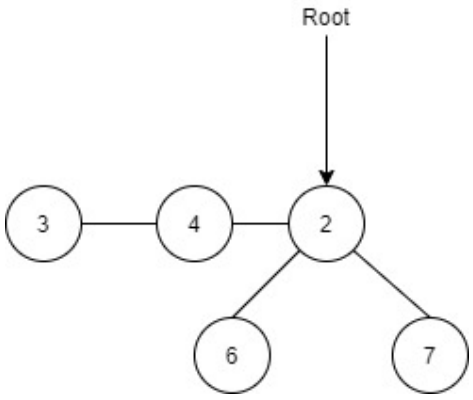
Now, take another example of the Fibonacci heap and you want to insert element 4.



Space for learners:

Space for learners:

After inserting 4, the Fibonacci heap will be as follows.

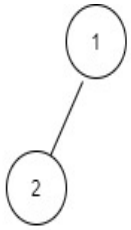


2.11 UNION OF FIBONACCI HEAP

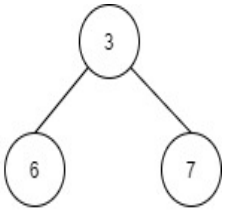
Union of the Fibonacci heap will be carried out by using the following steps.

i) You need to concatenate the roots of both Fibonacci heaps.

ii) Then update the min by selecting the minimum key from the root list.

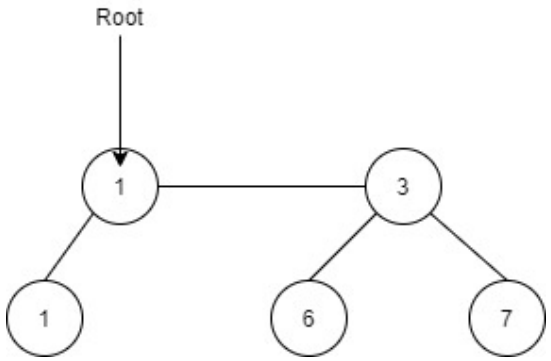


Fibonacci heap 1



Fibonacci heap 2

In the above diagram, you have two Fibonacci heaps i.e. Fibonacci heap 1 and Fibonacci heap 2. Now you can add these two heap by simply extracting min from the heaps and pointed it. After pointing the root, concatenate the heap.

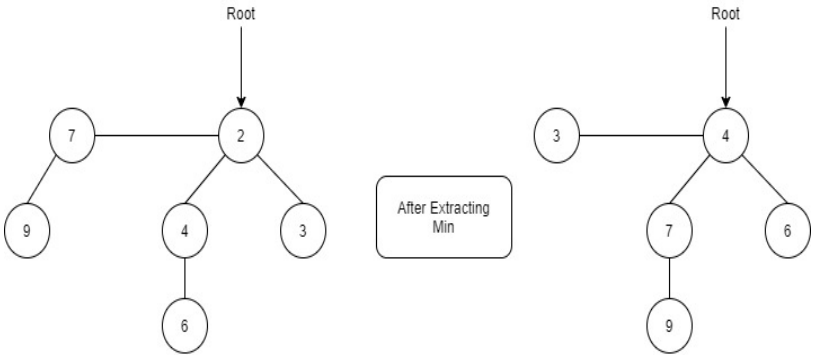


Space for learners:

2.12EXTRACT MIN FROM FIBONACCI HEAP

For finding the minimum, you need a function for deleting the minimum node and set the min pointer to the minimum value in the remaining heap. The following steps are followed:

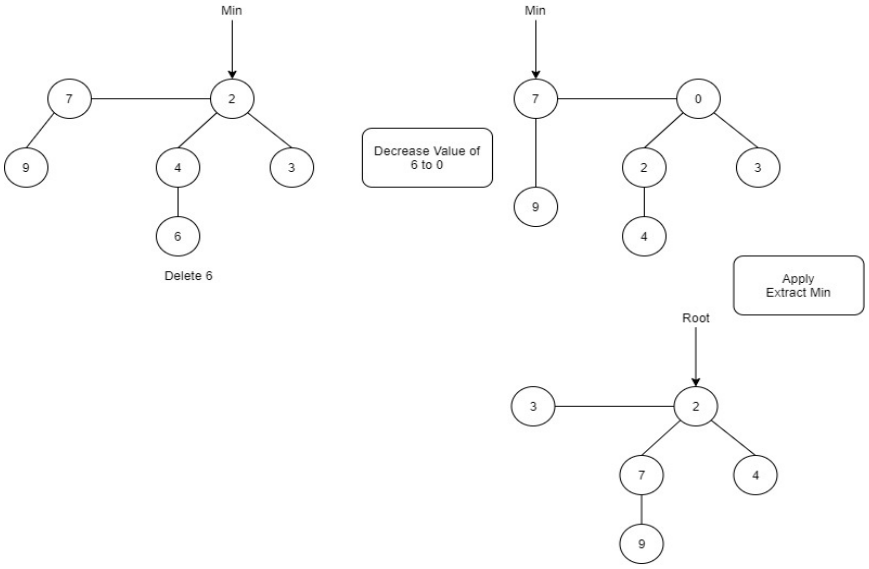
1. Delete the min node from the heap.
2. Now, point the head to the next min node and add all the trees of the deleted node in the root list.
3. Create an array of degree pointers of the size of the deleted node.
4. Set degree pointer to the current node.
5. Move to the next node.
   - If degrees are different then set degree pointer to next node.
   - If degrees are the same then join the Fibonacci trees by union operation.
6. Repeat steps 4 and 5 until the heap is completed.



2.13 DELETE A NODE IN FIBONACCI HEAP

To delete a node from the Fibonacci heap, the following steps are followed:

1. Decrease the value of the node to be deleted ‘n’ to a minimum by Decrease_key() function.
2. By using the min-heap property, heapify the heap containing ‘n’, bringing ‘n’ to the root list.
3. Apply Extract_min() algorithm to the Fibonacci heap
4.



Space for learners:

2.14 AMORTIZED ANALYSIS

Amortized analysis is applied to data structures that support many operations. The classical asymptotic analysis gives a worst-case analysis of each operation. Anamortized analysis focuses on a sequence of operations, an interplay between operations, and thus yielding an analysis that is precise and depicts a micro-level analysis. The characteristics of the amortized analysis are given below.

i) It is applicable where an occasional operation is very slow, but most of the other operations are faster.

ii) In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a, particularly expensive operation.

iii) Amortized analysis is an upper bound.

iv) Amortized analysis may consist of a collection of cheap, less expensive, and expensive operations.

Amortized analysis of the Fibonacci heap takes the following running complexities which are less than the binomial heap.

a. Finding min will take Θ(1)
b. Delete min will take O(logn)
c. Insert element in the Fibonacci heap will take Θ(1)
d. Decrease Key will take Θ(1)
e. Merging will take Θ(1)

Check Your Progress-II

5. What is a Fibonacci heap?
6. What is amortized analysis ?
7. Why does amortized analysis important?
8. Define the complexities the binomial heap operation.
9. Define the complexities of the Fibonacci heap.

Space for learners:

2.15 SUMMING UP

Space for learners:

i) A heap tree is a binary tree where the root node is either maximum or minimum.

ii) The priority queue is discussed concerning the binomial heap. A binomial heap is a priority queue where the heap is merged.

iii) A binomial heap is a priority queue data structure that allows pairs of heaps to be merged. It is important because of the mergeable heap abstract data type.

iv) A binomial heap is a collection of binomial trees where each binomial tree is a min-heap tree. It has the following properties.

   a. Each binomial heap obeysthe properties of the min-heap.

   b. For K (non-negative) value, there should be atleast one binomial tree where the root has degree k.

v) The binomial heap is represented in memory. The first block of the memory represents the parent or root of the tree. The second block store the key values whereas the third block store the degree of the tree.

vi) If you want to create a binomial heap of 'n' nodes, then that can be easily defined by the binary number of the n. For example: if you want to create the binomial heap of 5 nodes. Then initially you have to find the binary number of the 5, and i.e., 101. From the binary number, you can observe that 1 is available at the 0 and 2, positions. Therefore, the binomial heap with 5 nodes will have a0 and a2binomial trees.

vii) Creating a binomial heaptakesO(1) time because creating a heap will create the head of the heap without any element.

viii) Finding the minimum key from a binomial heap means extracting the minimum value. Here you need to compare only the root node of all the binomial trees.

ix) To find the minimum key with a complexity value of O(logn).

x) Union of two binomial heaps means finding the union of two binomial heaps. The time complexity for finding a union is O(logn).

xi) Inserting a node in a binomial heap takes O(logn).

xii) Extracting the minimum key from the binomial heaptakesO(logn).

xiii) Deleting a node from the binomial heap takesO(logn).

xiv) To reduce the computational time, the Fibonacci heap is used.

xv) Finding min in Fibonacci Heap will take Θ(1)

xvi) Delete min from Fibonacci Heap will take O(logn)

xvii) Insert element in the Fibonacci heap will take Θ(1)

xviii) Decrease Key in Fibonacci Heap will take Θ(1)

xix) Merging in Fibonacci Heap will take Θ(1)

xx) Amortized analysis is applied to data structures that support many operations. The classical asymptotic analysis gives the worst-case analysis of each operation.

xxi) It is applicable where an occasional operation is very slow, but most of the other operations are faster.

xxii) In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time which is lower than the worst-case time of a, particularly expensive operation.

xxiii) Amortized analysis is an upper bound.

xxiv) Amortized analysis may consist of a collection of cheap, less expensive, and expensive operations.

Space for learners:

2.16 ANSWER TO CHECK YOUR PROGRESS

1. A binomial tree of order 0 has 1 node. A binomial tree of order 1 has 2 nodes whereas the 4 nodes are present if the order of the tree is 2. It has 2^k noes in the tree.

2. A binomial heap is a priority queue data structure that allows pairs of heaps to be merged. It is important because of the mergeable heap abstract data type.
3. i) True ii) True
4. Three binomial tree
5. Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can have any shape even all trees can be a single node.
6. Amortized analysis is applied to data structures that support many operations. The classical asymptotic analysis gives a worst-case analysis of each operation. An amortized analysis focuses on a sequence of operations, an interplay between operations, and thus yielding an analysis that is precise and depicts a micro-level analysis.
7. The classical asymptotic analysis gives the worst-case analysis of each operation. Anamortized analysis focuses on a sequence of operations, an interplay between operations, and thus yielding an analysis that is precise and depicts a micro-level analysis. This is the reason that the amortized analysis is important.
8. The complexities of the different operations of the binomial heap are
    a. To find the minimum key with a complexity value of O(logn).
    b. Union of two binomial heaps means finding the union of two binomial heaps. The time complexity for finding a union is O(logn).
    c. Inserting a node in a binomial heap takes O(logn).

Space for learners:

d. Extracting the minimum key from the binomial heaptakesO(logn).

e. Deleting a node from the binomial heap takesO(logn).

9. The complexities of the different operations of the Fibonacci heap are

   a. Finding min in Fibonacci Heap will take Θ(1)

   b. Delete min from Fibonacci Heap will take O(logn)

   c. Insert element in the Fibonacci heap will take Θ(1)

   d. Decrease Key in Fibonacci Heap will take Θ(1)

   e. Merging in Fibonacci Heap will take Θ(1)

Space for learners:

2.17 POSSIBLE QUESTIONS

Short answer type questions:

i) What is a binomial heap?

ii) Why does binomial tree be important to construct the binomial heap?

iii) How many binomial trees will be required to construct a binomial heap of node 13? What is the degree of the heap?

iv) How does union perform in a binomial heap?

v) How does union perform in a Fibonacci heap?

vi) What is the difference between the operations of the Fibonacci and binomial heap

vii) What is the amortized analysis?

viii) Can we perform amortized analysis in a binomial heap?

Long answer type questions:

i) Explain the different operations of the binomial heap with diagrams and complexities.

ii) Explain the different operations of the Fibonacci heap with diagram and complexities

Space for learners:

2.18 REFERENCES AND SUGGESTED READINGS

- Karumanchi, Narasimha. *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*. almohreraladbi, 2011.
- Thareja, Reema. Data structures using C. Oxford University Press, Inc., 2011.
- Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2022.

---×---

UNIT 3: PARTITION ADT

Unit Structure:

Space for learners:

3.1 INTRODUCTION

This unit presents the union-find algorithm in detail. The union-find algorithm data structure is defined and its uses in terms of disjoints sets are explained. Along with the aforesaid mention, the unit also discusses the procedure of finding the cycle in an undirected graph. The union by rank and path compression is also discussed in this unit. Again optimized path compression is also discussed in this unit.

3.2 UNIT OBJECTIVES

After going through this unit, you will be able to know

i) About union-find data structure.

ii) About disjoints sets and union-find.

iii) About union by rank and path compression.

3.3 UNION FIND DATA STRUCTURE

Union Find data structure that stores a collection of disjoint sets. It stores a partition of set into disjoints subsets. It allows to add new subsets and merge them i.e. union. A union finds data structure generally reduces the time of execution by dividing the elements into subsets.

The data structure tells its definition itself. The FIND means to search for something in a set. The UNION means to join something. Let's, two subsets are given as below.

A = {1, 3, 5, 7}

B = {2, 4, 6, 8}

So in this case the Find (2) = A. It means that the value 2 is present in set A. Again, Find (6) = B. It means that 6 are present in set B. If Find (A)=Find (B), then it is considered that the elements are present in the same set.

For Union operation, the two sets are joining together. UNION (A, B) = {1.2.3.4.5.6.7.8}. These two operations of the data structure are used to detect the cycle in an undirected graph.

Space for learners:

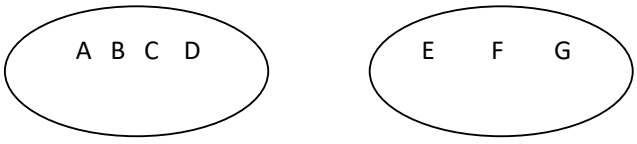
3.4 DISJOINTS SETS AND UNION FIND

A disjoint set data structure is a data structure that deals with the element partitioned into different disjoint subsets. Lets you have a S1 = {1, 2, 3, 4} and S2 = {5, 6, 7, 8}. So, these two sets are known as disjoint sets because the common of these two sets is Ø. Along with the common set, the union operation of the two sets gives S = {1, 2, 3, 4, 5, 6, 7, 8}. It means that the union-find algorithm performs two operations on disjoint data structure, i.e. ***Find*** and ***Union***. A disjoint data structure is sometimes called a union-find data structure or merges data structure.

i) ***Find:*** In this operation, an element is searched and determine swhich subset a particular element is in.

ii) ***Union:*** It is used to find the union of two sets.

For better understand, lets you have two sets.



Set A Set B

In the above, two sets are there-Set A has 4 elements i.e. A, B, C, and D. Set B to have three elements likely E, F and G. Now, these two sets are disjointed because their intersection is null. If the union

operation is performed in between sets, then the union is U = {A, B, C, D, E, F, G}. After applying the Find (A), Find (G), it returns the set U. Because A and G are present at U.

The disjoint data structure is used to check whether a nondirected graph contains a cycle or not. For this, Union-Find Algorithm is used using an array representation. The pseudo code to detect the cycle in the undirected graph using the union-find algorithm is given below.

For each unvisited edge (u,v) in the edge set E

```
{
        if(find(u) = = find(v))
        {
                Cycle detected;
        }
        else
        perform Union(u,v);
}
```

Let's have a graph of five vertices as shown in the diagram.

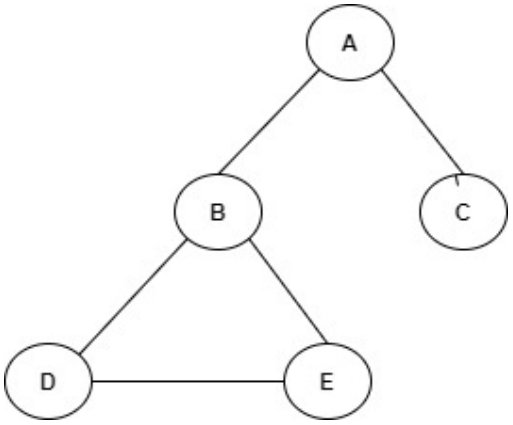


Figure 3.1: A graph with 5 vertices.

From the vertices, you should form a single array called a parent and its indices are represented by the vertices.

-1	-1	-1	-1	-1
A	B	C	D	E

Space for learners:

The individual value inside the array parent [] is -1 and it denotes that each vertex is present in its own set and its parent is itself i.e. the parent of A is A. Now you need to update the value of the array according to the edge of the graph. Let's consider edges one by one.

i) For the First edge, A-B. The value present for A in the array is -1, i.e., Find(A) = A. It means that the parent of A is A. Again, the value of B is -1, i.e., Find(B) = B. It means that these two nodes are in different sets. So, union operation is required according to the algorithm. After union operation AB = {A,B}. So, parent array is required to present these two values. Here, the B will act as a parent of A and the set denoted by the B. The value of the A in the array changes from -1 to 1.

B	-1	-1	-1	-1
A	B	C	D	E

ii) Consider the 2nd edge, i.e. AC. Here the Find(A)=B, and Find(C)=C. As they belong to two different sets, so union operation is required to perform, i.e., Union(A, C). After union, the parent of B is C.

B	C	-1	-1	-1
A	B	C	D	E

iii) The next edge is BD. Find(B) = C, Find(D)=D. So do union operations using Union(B, D). After union the parent of C is D.

B	C	D	-1	-1
A	B	C	D	E

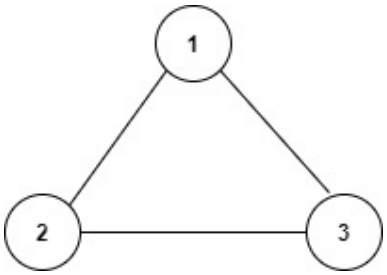
iv) Take the edge BE. Find(B)= C and Find(E)=E. Do union using Union(B,E). The parent of D is E.

B	C	D	E	-1
A	B	C	D	E

v) Now take the edge DE, Find(D) = E, Find(E)=E. It means that the two values are the same. So, there is a cycle.

Space for learners:

For example- Consider the following undirected graph. It contains 3 vertices as follows.



For the Find-Union Algorithm, consider the following array.

1	2	3
-1	-1	-1

Initially, the value of parent of each node is -1. It means that the parent of each node is itself. Now apply the following steps.

i) Consider the edge 1-2. Find(1) = 1, Find(2) = 2, They are not same. So, do union using Union(1,2). After performing union the parent of 1 will be 2.

1	2	3
2	-1	-1

ii) Now consider the next age 2-3. Find(2) = 2, Find(3) = 3. Do apply Union(2,3) and it will produce {1,2,3} After union the parent of 2 is 3.

1	2	3
2	3	-1

iii) Now consider the edge 1-3. Find(1)=2 and Find(3)=3. It means that they are in the same set. So there is a cycle in the undirected graph.

Space for learners:

CHECK YOUR PROGRESS - I

1. What is a disjoint set?
2. What is ADT? Is disjoint set ADT?
3. What is Union Find Algorithm?
4. True or False
   i) Initially each vertex represents itself as a parent in Union Find.
   ii) The Worst-case complexity of Union find is O(n).

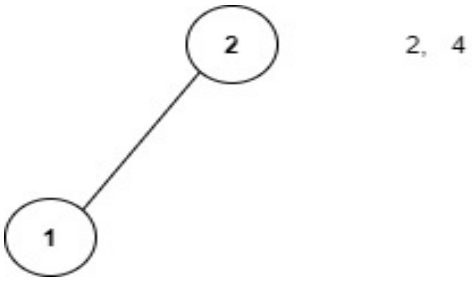
Space for learners:

3.5UNION FINDRANK AND PATH COMPRESSION

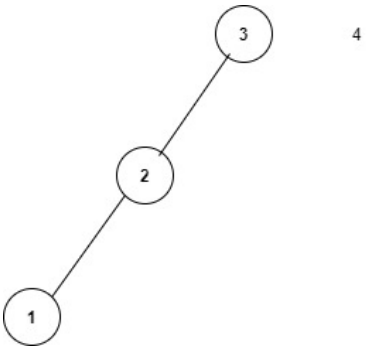
In section 3.2, the details of the union-find algorithm to detect the cycle in a directed graph are discussed. The Union () and Find () takes O(n) times in the worst case. This can be updated using the Union rank algorithm. Let's understand by considering an array A = {1,2,3,4}

i) Initially, all the elements are in a single element set.

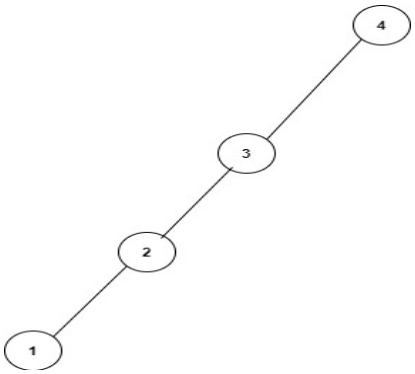
ii) Do perform union in between 1 and 2 as follows.



iii) Now perform unions 2 and 3 as follows.



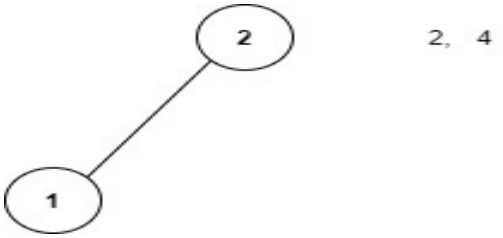
iv) Now perform union in between 3 and 4 as follows.



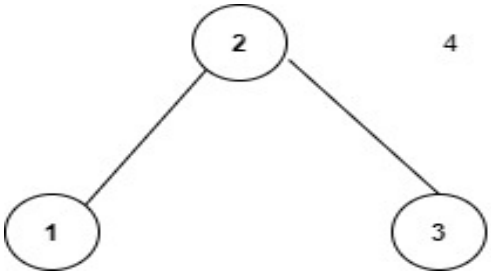
The above operation takes O(n) times but it can be optimized to O(logn) using the union rank algorithm. The union by rank always attached a smaller depth tree under the root of the deeper tree. In this technique, instead of finding the height, path compression is used and then the rank is not equal to height. The size of the tree can also be used as a rank. Let understand the union rank for the above example.

i) Initially, all the elements are in a single element set.

ii) Do perform union in between 1 and 2 as follows.

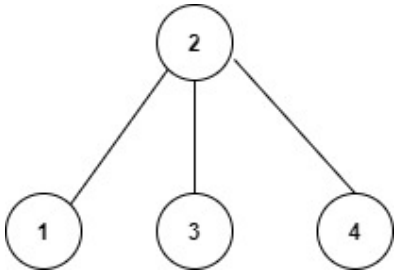


iii) Now perform unions 2 and 3 as follows.

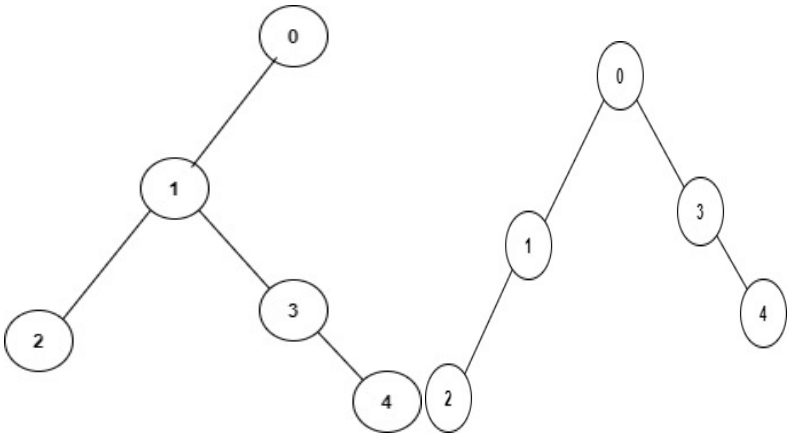


Space for learners:

iv) Now perform union in between 3 and 4 as follows.



Apart from the Union Rank algorithm, path compression is also used to optimize the time complexities in the worst case. It is done by flattening the tree at the time of the Find() algorithm call and it returns root. The Find() operation traverses up from a node 'n' to find the root. The concept of path compression is to find the root as the parent of 'n' so that you don't have to traverse all intermediate nodes again. If x is the root of a subtree, then path (to root) from all nodes under x also compresses. For example, after performing, Find (3), the tree will be changed as follows.



(a) (b)

Figure 3.2: a) Before Path Compression b) After Path Compression

The pseudo code for union by rank and path compression is given below.

A. UnionbyRank(m, n)

i. mRoot= Find(m)
ii. y=nRoot= Find(n)
iii. // if m and ynare already in the same set

Space for learners:

a. if mRoot == nRoot
b. return

iv. // m and n are not in same set, so we merge them
a. if mRoot.rank<nRoot.rank
i. mRoot.parent = nRoot
b. else if mRoot.rank>nRoot.rank
i. mRoot.parent = nRoot
c. else
i. mRoot.parent = nRoot
ii. mRoot.rank := nRoot.rank + 1

B. FindbyPathCompression(m)

i. if m.parent != m
ii. m.parent := Find(m.parent)
a. return m.parent

CHECK YOUR PROGRESS - II

5. What is a Union by rank?
6. What is Union by Path compression?
7. What are the worst-case complexities of the Union by rank and path compression algorithm?

3.6 SUMMING UP

1. Union Find data structure that stores a collection of disjoint sets. It stores a partition of set into disjoints subsets. It allows to add new subsets and merge them i.e., union.
2. A union finds data structure generally reduces the time of execution by dividing the elements into subsets.
3. The data structure tells its definition itself. The FIND means to search for something in a set. The UNION means to join something.
4. A disjoint set data structure is a data structure that deals with the element partitioned into different disjoint subsets.
5. A disjoint data structure is sometimes called a union-find data structure or merges data structure.

Space for learners:

i) ***Find:*** In this operation, an element is searched and determines which subset a particular element is in.

ii) ***Union:*** It is used to find the union of two sets.

6. The pseudocode to detect the cycle in the undirected graph using the union-find algorithm is given below.

```
For each unvisited edge (u,v) in the edge set E
{
    if(find(u) = = find(v))
    {
        Cycle detected;
    }
    else
    perform Union(u,v);
}
```

7. The union () and find () takes O(n) times in the worst case. This can be updated using the Union rank algorithm.
8. Apart from the Union Rank algorithm, path compression is also used to optimize the time complexities in the worst case.
9. The pseudo code for union by rank is given below.

Union by Rank (m, n)

i. mRoot = Find(m)
ii. y=nRoot = Find(n)
iii. // if m and ynare already in the same set
    a. ifmRoot == nRoot
    b. return
iv. // m and n are not in same set, so we merge them
    a. if mRoot.rank<nRoot.rank
        mRoot.parent = nRoot
    b. else if mRoot.rank>nRoot.rank

Space for learners:

mRoot.parent = nRoot

c. else

i. mRoot.parent = nRoot

ii. mRoot.rank:= nRoot.rank + 1

10. The pseudocode for union path compression is given below

Find by Path Compression(m)

i. if m.parent != m

ii. m.parent:= Find(m.parent)

b. return m.parent

Space for learners:

3.7 ANSWERS TO CHECK YOUR PROGRESS

1. A disjoint set data structure is a data structure that deals with the element partitioned into different disjoint subsets.
2. Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction. Yes Disjoint Set is ADT.
3. The union-find algorithm performs two operations on disjoint data structure, i.e. ***Find*** and ***Union***. A disjoint data structure is sometimes called a union-find data structure or merges data structure.
   - ***i)*** ***Find:*** In this operation, an element is searched and determines which subset a particular element is in.
   - ***ii)*** ***Union:*** It is used to find the union of two sets.
4. i) True ii) True
5. The union by rank is an algorithm to detect the cycle in an undirected graph at O(logn) time which is less than the traditional Union Find Algorithm.

6. The union by path compression is also an algorithm to detect the cycle in an undirected graph at O(logn) time which is less than the traditional Union Find Algorithm.
7. The worst-case time complexity of Union Rank and Union Path Compression is O(logn).

Space for learners:

3.8 POSSIBLE QUESTIONS

Short answer type questions:

i) What are disjoint sets? Give example.

ii) Can you merge a disjoint set in O(logn) time?

iii) What is the necessity of the Union-Find algorithm?

iv) What is the difference between Union-Rank and Union-Path Compression algorithms?

v) Which algorithm is best to find the cycle in an undirected graph?

vi) What is A ADT?

vii) Can you mention the disjoint set as ADT? If yes, why?

Long answer type questions:

i) Explain the union rank and union path compression algorithm with an example.

ii) Explain the Union-Find algorithm with an example.

iii) Show the complexities of Union-Rank and Union-Path Compression with their pseudocode.

3.9 REFERENCES AND SUGGESTED READINGS

- Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.
- https://www.geeksforgeeks.org/disjoint-set-data-structures/
- https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/

UNIT 4: B TREE AND B+ TREE

Unit Structure:

Space for learners:

4.1 INTRODUCTION

This chapter gives an overview of the B tree and B+ tree. A B tree is a data structure that is used for external memory indexing. The different operations with examples are explained and show in this unit. The different cases of insertion and deletion of keys from the B tree are demonstrated in this unit. The difference between the B tree and the B+ tree is also shown in this unit. The properties, operations, and complexities of the B+ tree are discussed in this unit. The insertion and deletion cases with proper examples are explained in this unit.

4.2 UNIT OBJECTIVES

After going through this unit, you will be able to know

i) About B tree and its properties.

ii) About the different operations of the B tree.

iii) About B+ tree and its properties.

iv) About the different operations of B+ tree

Space for learners:

4.3 B TREE

When a large amount of data is there in a self-balancing tree, others self-balancing tree-like AVL, Red-Black tree considers that elements are in the main memory that is practically not possible. In this case, data are read from an external disk in the form of a disk. But disk access time is high than the main memory access time. So, to reduce the disk access time, B Tree is used. Because, it allows to perform search, insertion, and deletions of the element in logarithmic time.

A B-tree is a data structure that is usually used in database and file structure. It is known as a balanced m way tree where m is an order. It is also known as the generalization of the B tree. A B-Tree of order m can have at most m-1 keys and m children. It has more than two children with the following properties.

- The leaves of a B tree are at the same level.
- A minimum degree 'm' is used to represent the degree of the tree (nos. of children).
- The tree may have a maximum of m children and a minimum of [m/2] children.
- Every node except the root of a B tree contains at least [m/2-1] keys. The root may contain a minimum of 1 key.
- All nodes (including root) of a B Tree may contain at most m - 1 key.
- The root nodes must have at least 2 nodes. The internal should have [m/2] children whereas the leaf has 0 children.
- All leaf nodes must be at the same level.

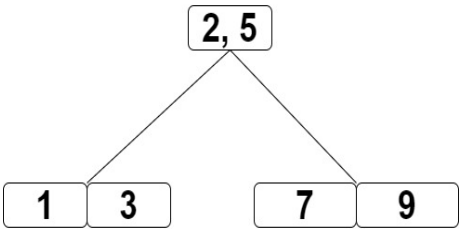


Figure 4.1: Example of B Tree.

4.4 OPERATIONS OF B TREE

Space for learners:

A B-tree has the following operations.

i) Search operation

ii) Insert Operation

iii) Delete Operation

i) Search Operation of B Tree

The simplest operation of B Tree is the search operation. The following steps are considered for the search operation in B Tree

I. Let the key (the value) be searched by "K". The searching operation starts from the root and recursively traverses down.

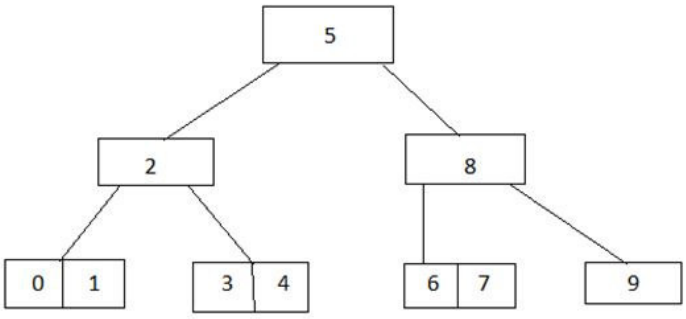
II. If the key (K) is lesser than the root value, start to search the left subtree. If K is greater than the root value, start to search for the right subtree.

III. If the search element and key element match, simply return the node.

IV. If the K is not found in the node, traverse down to the child with a greater key.

V. If k is not found in the tree, return NULL.

Let's you have the following B tree and you are searching for K = 7.



The searching operation can be executed as follows.

i) Initially Start your search from the root. The root is 5. It is not matching with the key. The key 7 is more than the

5. So, in the next step, start to search for the right subtree.

ii) On the right side, the element is 8 which is not matched with the key 7. But the key is lesser than the 8. So, traverse left for searching.

iii) The next node contains two elements 6 and 7. The node value is 7 matches the key value. The search is successful.

ii) Insertion Operation of B Tree

The insertion of a key in a B tree requires the first traversal in B-tree. There are two cases for inserting the key that are:

i) If the leaf node in which the key is to be inserted is not full, then the insertion is done in the node.

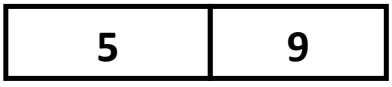
ii) If the node were to be full then insert the key in order into an existing set of keys in the node, split the node at its median into two nodes at the same level, pushing the median element up by one level.

Let's understand the insertion of elements in the B tree of minimum order m =3. The elements to be inserted in the B tree are 5,9,3,7, 1 and 2.

Let's consider that the tree is empty.

i) As the tree is empty, insert the 5.

ii) The root contains a maximum of 2 elements like the order of the tree is 3. So, elements 9 also can be inserted in the root node. But insertion should take place in a sorted manner. As 9 is greater than the 5, so it will be right of 5 in the same node.

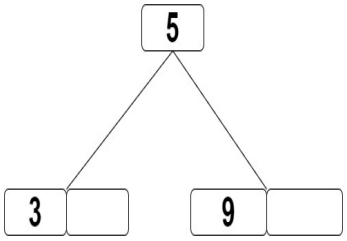


iii) The next element is 3. But the first node is full, so insertion is possible in the node. Because 2 nodes are possible to insert in any node. So, the element 3 will be added to the next level after splitting the first node. Before splitting, lets 3 is inserted in the first node which

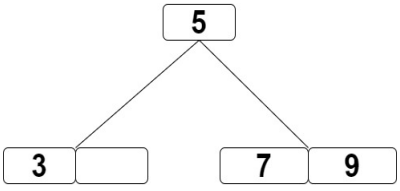
Space for learners:

is not possible. So, split it by finding the median element. In that case, 5 will go up and acting as the root of 3 and 9.

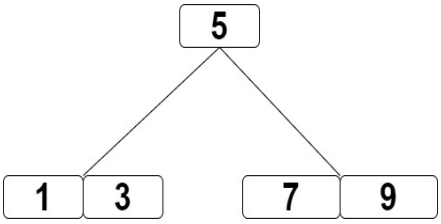
So the elements are rearranged as follows.



iv) Now the net element is 7. So, start comparing from the root. The 7 is greater than 5, so it will go right of 5. As 7 is lesser than 9, it cannot be the right of 9. So, after rearranging the tree will be as follows.



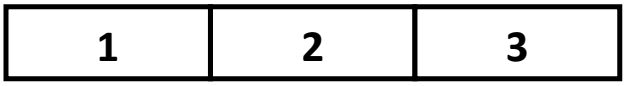
v) The next element is 1. Again compare 1 and root key. The element is lesser than 5, so it will be placed at the left of 5. After changing the positions of 1 and 3 for sorting, the tree is as follows.



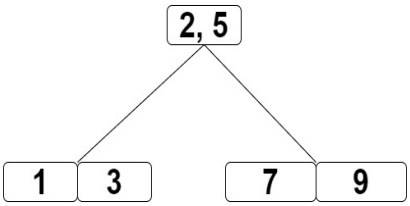
vi) The next element is 2. Again, compare with root 5. It is lesser than root, so mode to the left of root. But the left node of the root is already filled up. So, splitting is

Space for learners:

required to insert node 2. Let's have inserted 2 in the node as follows which is not possible.



In this situation, node 2 will go up and it will be combined with the root to make the final B tree.



The same should follow for all the elements that you need to insert in a B-tree by maintaining the property of the B Tree.

iii) Deletion Operation of B Tree

In deletion operation, a key element can be considered as the target key. There are two possibilities.

i) The target may be present in the leaf node

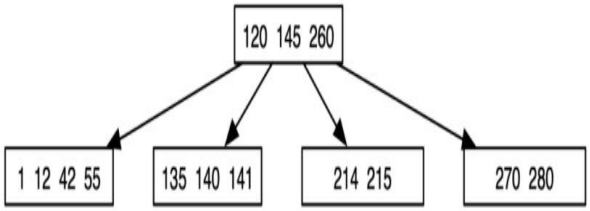
ii) The target may be present in the internal node.

Now consider the case [i], i.e., the target is present in the leaf node. Again there are two possibilities.

a) The leaf node may have more than a minimum number of keys.

b) The leaf node contains the minimum number of keys.

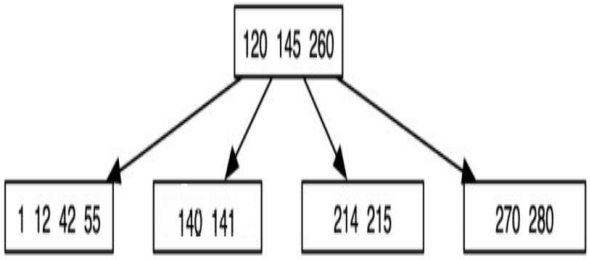
For better understand, consider the following B Tree of order 5.



Space for learners:

In the above tree, the minimum number of keys in the tree is [m/2-1] = 2, and the maximum number of keys in the tee is [m-1] = 4. So, let's consider the first possibility of a leaf node and the target element is 135 and

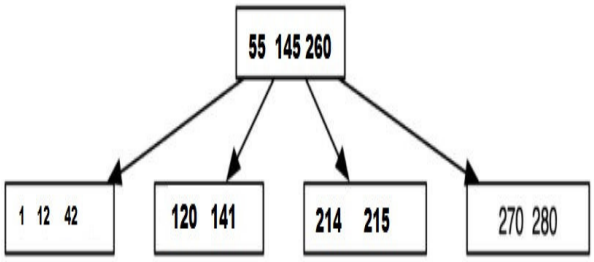
i) First, search element 135. It is in the second leaf node. The second leaf node contains the three keys. So, it is more than the minimum number of keys. In this possibility, one can simply delete the key as it will not violate the property of the B tree. So, delete node 135.



ii) In the above B tree, all the node contains minimum 2 keys. It means that it follows the properties of the B Tree. Now delete node 140. If you delete node 140. It will violate the rule of the B tree because that node will contain only one key. In this case, the node will borrow a key from the sibling node (Left or right). But borrowing the node from a sibling is possible only through the parent node. Let discuss the issues. If you delete node 140 from the node, it will borrow the maximum key 55 from its left sibling. After borrowing one node from the left sibling, the left sibling still contains 3 keys and it will not violate the properties of the B tree. But borrowing the key 55 will happen through the root. The first 55 will go to the root as [55, 120, 145, 260] and the root node will send the root of the key of 55 and [140, 141], i.e., 120 to the 2nd leaf node. At the same time key, 140 is deleted from the leaf node.

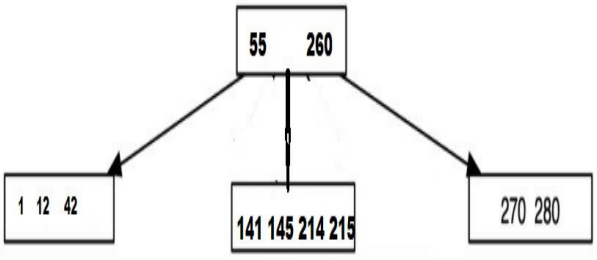
Space for learners:

Space for learners:



In some cases, the borrowing may not be possible from the left sibling. In that case, borrowing may happen from the right siblings by following the same process as mentioned above.

Now delete the key 120 from the B tree. If key 120 will be deleted directly, it will violate the properties of the B tree. In this situation, merging can be done to validate the B tree. But the merging will happen through the root key of the 1st and 2nd leaf node of the tree, i.e. 55. By consider the 55, the merging node will contains [1, 12, 42, 55, 120, 141]. If one can delete the key 120 from the merging node, still it contains 5 keys, i.e. [1,12,42,55,141]. So, left merging is not possible and one needs to perform the right merging. The root key of the 2nd and 3rd leaf nodes is 145. So, by joining the key root, the merging node will be [120, 141, 145, 214, 215]. Now delete key 120, and the merging node will be [141, 145, 214, 215]. It will not be violating the properties of the B tree.



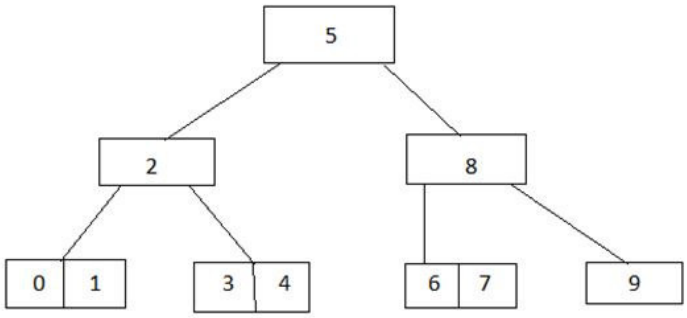
Now consider that the target key is present in the internal node. If the key to be deleted lies in the internal node, the following cases occur.

i) The internal node, which is deleted, is replaced by an in order predecessor if the left child has more than the minimum number of keys.

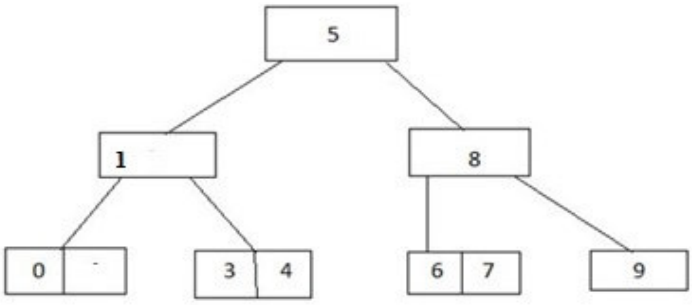
ii) The internal node, which is deleted, is replaced by an in order successor if the right child has more than the minimum number of keys.

iii) If either child has exactly a minimum number of keys then, merge the left and the right children.

Let's consider the following B tree of order 3 and delete node 2.

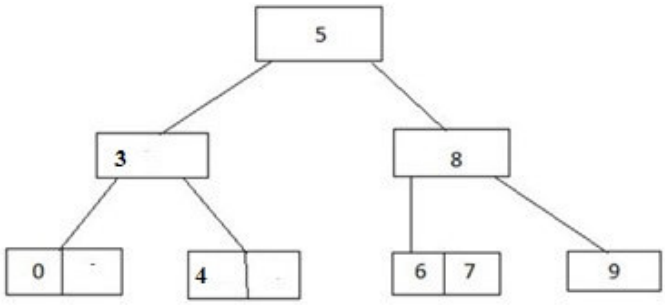


In this case of deleting key 2 from the internal node, the left child of the internal node has more than the minimum number of keys. So, it will follow the rule [i] as mentioned above. After deleting key 2, the predecessor of key 2 is 1. So, the key 2 will be replaced by 1 in the B tree and it will not violate the rule of the B tree.



If you again delete node 1 from the tree, it will be replaced by the in order successor of 1, i.e. 3.

Space for learners:



Space for learners:

Stop to Consider

- The minimum height of the B-Tree with n number of nodes and m is the maximum number of children of a node is: $\mathrm{h(min)} = [\log(\mathrm{m}+1)] - 1$
- The maximum height of the B-Tree with n number of nodes and d is the minimum number of children that a non-root node is: $\mathrm{h(max)} = [\log_t(n+1)/2]$ where $\mathrm{t} = \mathrm{m}/2$.

CHECK YOUR PROGRESS - I

1. What is a B tree?
2. What are the properties of the B tree?
3. Which is the most widely used external memory data structure?
4. B-tree of order n is a order-n multi way tree in which each non-root node contains _____
5. A B-tree of order 4 and of height 3 will have a maximum of _____ keys.
6. Five node splitting operations occurred when an entry is inserted into a B-tree. Then how many nodes are written?

4.5 B+ TREE

B+ Tree is an extension of B Tree that allows the efficient insertion, deletion, and search operations. It is a self-balancing tree that allowed to insert of the element or key in the leaf node only whereas the B tree is allowed to insert in the leaf and internal nodes. The leaf nodes of a B+ tree are linked together in the form of a singly linked list to search queries more efficiently. A large amount of data can be stored in a B+ tree that cannot be stored in the main memory. The size of the main memory is always limited and due to this reason, the internal nodes of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The properties of a B+ Tree are as follows.

i) All the leaves are at the same level.

ii) The root has atleast 2 children.

iii) Each node except the root node has atleast m/2 children and a maximum of m children.

iv) Each node of a B+ tree must contain a maximum of m-1 keys and a minimum of [m/2-1] keys.

4.6 B+ TREE OPERATIONS

Like the B tree, the following operations are also present in the B+ Tree.

i) B+ Tree insertion

ii) B+ Tree deletion

iii) B+ Tree searching

Let's discuss the B+ tree insertion. Let's following items are there to insert on the B+ tree of order 4. Elements = [1,4,7,10,17,21,31,25,19]. The insertion can happen in the following steps.

i) As the tree will be in order 4, so the following points should be noted before element insertion

   a) The maximum children = m = 4

   b) The minimum children = m/2 = 2

   c) The minimum number of key in the tree is- [m/2-1] = 1

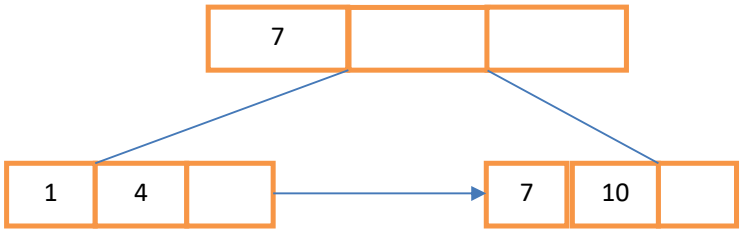
Space for learners:

d) The maximum number of key in a node is- m-1 = 3.

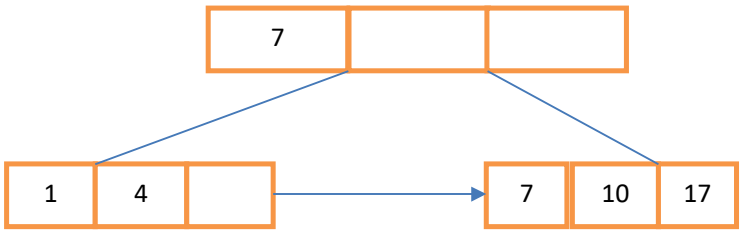
ii) Take the first element 1 and insert the first node where you can insert maximum of 3 keys.

iii) The second and third elements (4,7) are also inserted in the first node as below.

iv) Now to insert key 10, a node will be required as 10 cannot be inserted in the first node because it is already filled up. So, splitting will be required in this situation. On splitting 7 will go up and 1,4 will be present in the left leaf node and 10 will be present in the right leaf node with node 7. Node 7 will be present in the root node as well as in the leaf node because it is a B+ tree. In the B+ tree, each node should be present in the leaf node. The leaf node is connected by a link as follows.

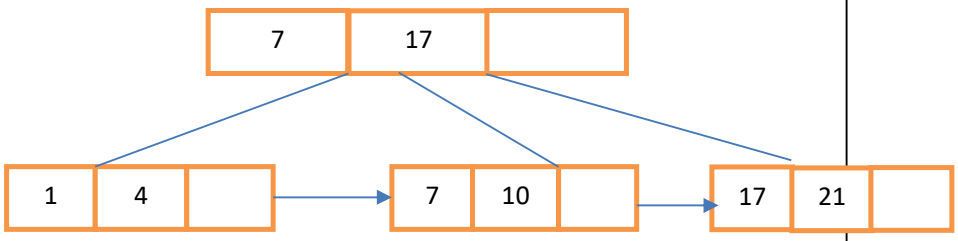


v) The next key is 17. It can be inserted as the right of 10 in the same node. [For insertion, start comparing new elements from the root. If it is more than root, go to the right of the root node, else left].

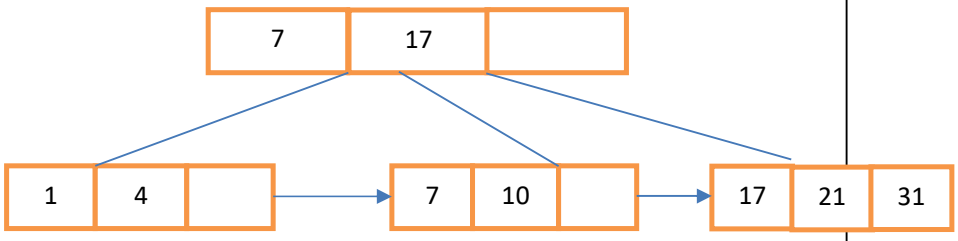


Space for learners:

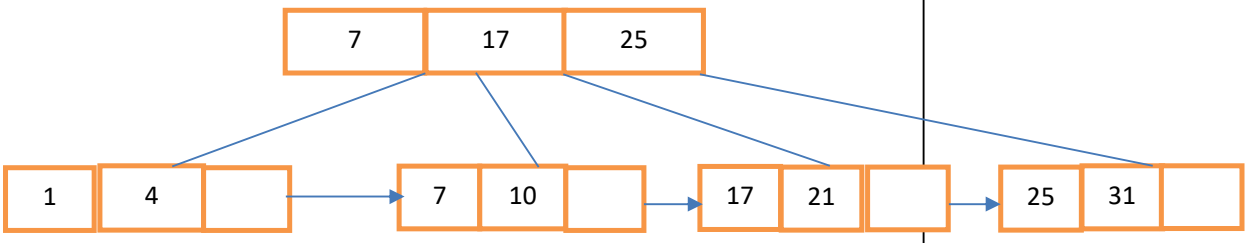
vi) The next key is 21. It will come to the right of 17 that is not possible as it is already filled up. So, splitting is required. After splitting the element 17 will go up and [7, 10] will be in left and [17, 21] will be the right leaf node. Here also 17 will present in the leaf node though it present in the root as an index.



vii) Next is 31. The 31 will be added as a right of 21.

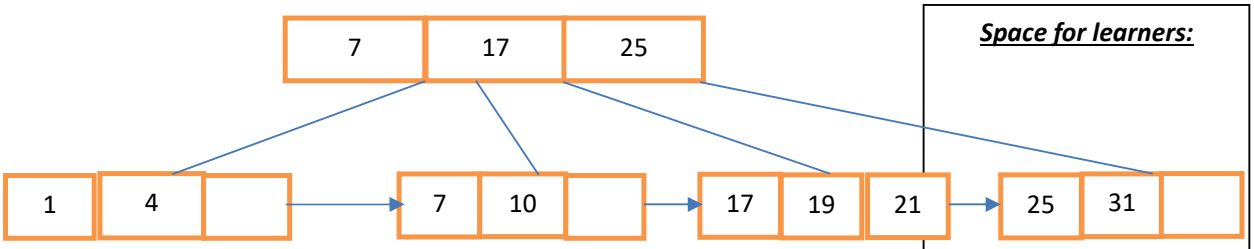


viii) The next key is 25. As a rule, 25 will come as a right of 21. But it is not possible as the node has been filled up. So, split it. After splitting 25 will go up and [17,21] will be present in the left node and [25,31] will present in the right node as follows.



ix) The next is 19 and it is added as the right of 17 as follows.

Space for learners:



Space for learners:

The deletion operation of the B + tree is the same as with the B tree. Let's discuss the deletion of the B+ tree with the following example.

Let, the key to be deleted from the B+ tree is present only at the leaf node, not in internal nodes. There are two cases for it:

i) If there is more than the minimum number of keys in the leaf node then simply delete the key.

ii) If there is an exact minimum number of keys in the node then delete the key and borrow a key from the left or right sibling. Add the median key of the sibling node to the parent.

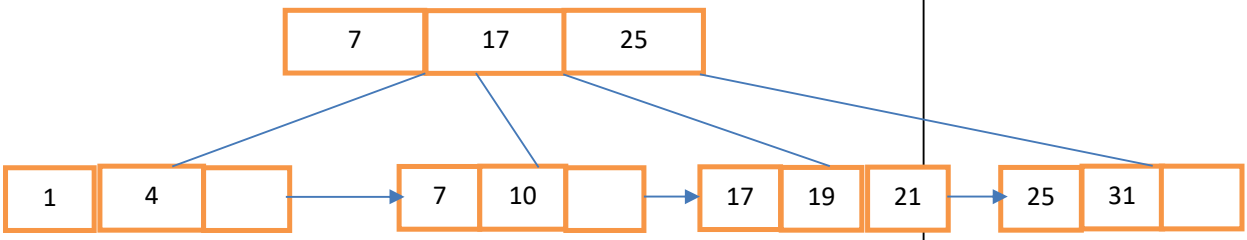
Let's consider the following B+ tree of order 4. As the tree will be in order 4, so the following points should be noted before element deletion.

a) The maximum children = m = 4

b) The minimum children = m/2 = 2

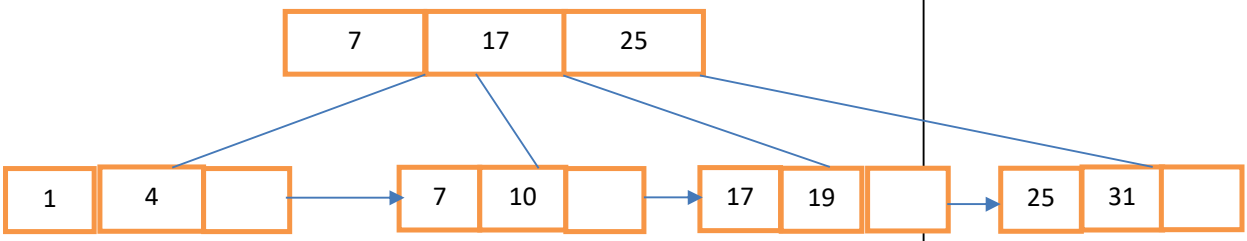
c) The minimum number of key in the tree is [m/2-1] = 1

d) The maximum number of key in a node is m-1 = 3.

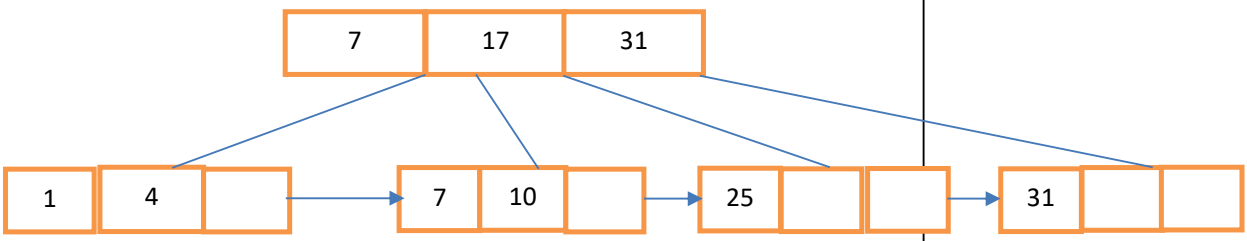


First, delete 21. In the 3rd leaf node, the total number of keys is 3 that is more than the minimum number of keys in a node. So, simply delete node 21. The B+ tree after deleting node 21 is as follows.

Space for learners:

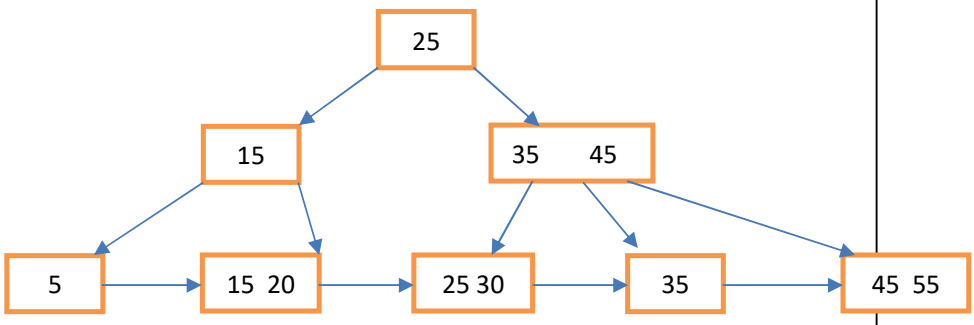


Now again delete 17 and 19. First, delete 19. As the nodes have two keys, so simply delete node 19. After deleting 19, the node contains only 17, i.e., case ii of condition 1. So, delete node 17, but borrow a key from the right sibling. After borrowing, 31 will go up and 25 will come as left of 31 in the left leaf node as follows.



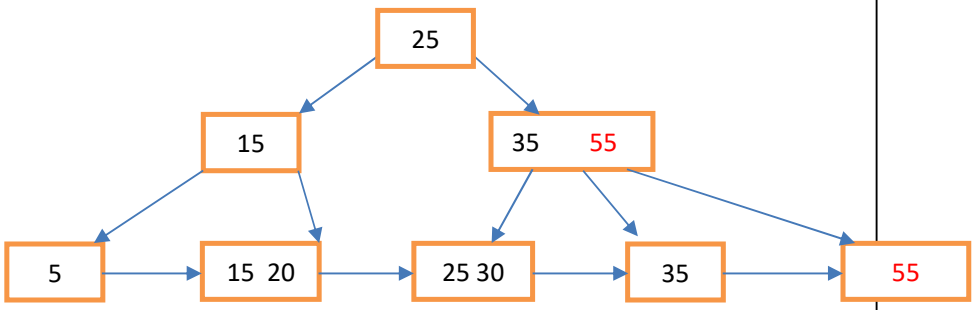
Sometimes the key to be deleted is present in the internal nodes as well. Then you have to remove them from the internal nodes as well. There are the following cases for this situation.

e) If there is more than the minimum number of keys in the node, simply delete the key from the leaf node. Then delete the same key from the internal node also. Fill the space in the internal node with the in-order successor.



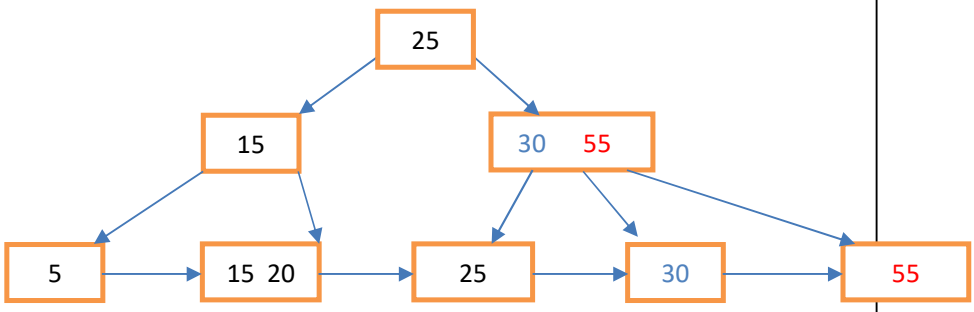
The above figure, lets you want to delete key 45. The node that contains 45, has more than the minimum number of keys. So simply

delete the key 45, and replace it within in order successor value i.e. 55.



iii) If the node contains the exact number of minimum keys then delete the key by borrowing a key from its immediate sibling (through the parent).

Let's you want to delete key 35. As the node contains only one element that is 25, one can delete key 35 by borrowing one key i.e. 30 from the left node. The key 30 will go up, the key 35 is deleted as follows.



4.7 EXTERNAL SORTING

When main memory can handle a large amount of data, external sorting is used for a class of sorting algorithms to handle massive amounts of data. It is required when data is not fit into the main memory to sort. It is a combination sort-merge technique. In the sorting technique, a chunk of keys is placed in the main memory and sort them. After that, the next chunk of data is placed main memory and then sort them again. Finally, all chunks are merged to sort all the elements. One of the external sorting is merge sort where a small chunk of elements are placed in a RAM and then sort them. Finally

Space for learners:

merge the resulting runs into successively bigger runs, until the file is sorted.

4.8 COMPLEXITIES OF B TREE AND B+ TREE

To reduce the disk access time, B Tree and B+ tree are used. Because, it allows to perform search, insertion, and deletions of the element in logarithmic time. So, the Complexities of the B tree and B+ tree are presented below.

Sr. No.	Operation	Algorithms	Time Complexity
1.	Search	B Tree	O(log n)
		B+ Tree	$O(t \log_t n)$
2.	Insert	B Tree	O(log n)
		B+ Tree	O(log n) O(M*log n + log L)
3.	Delete	B Tree	O(log n)
		B+ Tree	O(log n) O(M*log n + log L)

CHECK YOUR PROGRESS - II

7. What is a B+ tree?
8. A B+ tree can contain a maximum of 7 pointers in a node. What is the minimum number of keys in leaves?
9. B+ -tree has greater depth than corresponding B-tree (True or False)
10. B+ data structures are preferred in database-system implementation (True or False)
11. B + tree allows rapid random access as well as rapid sequential access (True or False)

Space for learners:

4.9 SUMMING UP

Space for learners:

1. A B-tree is a data structure that is usually used in database and file structure. It is known as a balanced m way tree where m is an order. It is also known as the generalization of the B tree. A B-Tree of order m can have at most m-1 keys and m children.
2. It has more than two children with the following properties.
    - The leaves of a B tree are at the same level.
    - A minimum degree 'm' is used to represent the degree of the tree (nos. of children).
    - The tree may have a maximum of m children and a minimum of [m/2] children.
    - Every node except the root of a B tree contains at least [m/2-1] keys. The root may contain a minimum of 1 key.
    - All nodes (including root) of a B Tree may contain at most m - 1 key.
    - The root nodes must have at least 2 nodes. The internal should have [m/2] children whereas the leaf has 0 children.
    - All leaf nodes must be at the same level.
3. A B-tree has the following operations.
    - Search operation
    - Insert Operation
    - Delete Operation
4. The simplest operation of B Tree is the search operation. The following steps are considered for the search operation in B Tree
    - Let the key (the value) be searched by "K". The searching operation starts from the root and recursively traverses down.
    - If the key (K) is lesser than the root value, start to search the left subtree. If K is greater than the root value, start to search for the right subtree.
    - If the search element and key element match, simply return the node.

- If the K is not found in the node, traverse down to the child with a greater key.
- If k is not found in the tree, return NULL.

5. B+ Tree is an extension of B Tree that allows the efficient insertion, deletion, and search operations. It is a self-balancing tree that is allowed to insert the element or key in the leaf node only whereas the B tree is allowed to insert in the leaf and internal nodes.
6. The properties of a B+ Tree are as follows.
   - All the leaves are at the same level.
   - The root has atleast 2 children.
   - Each node except the root node has atleast m/2 children and a maximum of m children.
   - Each node of a B+tree must contain a maximum of m-1 keys and a minimum of [m/2-1] keys.
7. The key to being deleted from the B+ tree is present only at the leaf node, not in internal nodes. There are two cases for it:
   - If there is more than the minimum number of keys in the leaf node then simply delete the key.
   - If there is an exact minimum number of keys in the node then delete the key and borrow a key from the left or right sibling. Add the median key of the sibling node to the parent.
8. When main memory can handle a large amount of data, external sorting is used for a class of sorting algorithms to handle massive amounts of data. It is required when data is not fit into the main memory to sort. It is a combination sort-merge technique.
9. The time complexity of search, insert and delete of the B tree is O(logn).The time complexity of search, insert and delete ofthe B+ tree are O(t $\log_t$n), O(log n) O(M*log n + log L) and O(log n) O(M*log n + log L).

Space for learners:

4.10 ANSWER TO CHECK YOUR PROGRESS

Space for learners:

1. A B-tree is a data structure that is usually used in database and file structure. It is known as a balanced m way tree where m is an order. It is also known as the generalization of the B tree. A B-Tree of order m can have at most m-1 keys and m children.
2. It has more than two children with the following properties.
    - The leaves of a B tree are at the same level.
    - A minimum degree 'm' is used to represent the degree of the tree (nos. of children).
    - The tree may have a maximum of m children and a minimum of [m/2] children.
    - Every node except the root of a B tree contains at least [m/2-1] keys. The root may contain a minimum of 1 key.
    - All nodes (including root) of a B Tree may contain at most m - 1 key.
    - The root nodes must have at least 2 nodes. The internal should have [m/2] children whereas the leaf has 0 children.
    - All leaf nodes must be at the same level.
3. B Tree
4. At least (n – 1)/2 keys
5. 255
6. 11
7. B+ Tree is an extension of B Tree that allows the efficient insertion, deletion, and search operations. It is a self-balancing tree that allowed to insert of the element or key in the leaf node only whereas the B tree is allowed to insert in the leaf and internal nodes. The leaf nodes of a B+ tree are linked together in the form of a singly linked list to search queries more efficiently.
8. 3
9. False
10. True
11. True

4.11 POSSIBLE QUESTIONS

Space for learners:

Short answer type questions:

i) What is a B tree?

ii) What is a B+ tree?

iii) What is the difference between B tree and B+ tree?

iv) What are the properties of the B tree?

v) What are the properties of a B+ Tree?

vi) What are the complexities of the different operations of the B tree and B+ tree?

vii) Insert the following element in a B tree Elements = [7,1,2,4,6,8,9,3]

viii) Why is the importance of pointer in the leaf node of a B+ tree?

ix) What is external sorting?

x) Is B tree an external memory technique? If yes, then why?

Long answer type questions:

i) Construct the B tree and B+ tree for the following elements.

   a) 1,5,2,3,8,15,41,14

   b) 2,5,41,24,23,7,8,9,12

   c) 1,2,3,4,5,6

ii) Construct and delete elements from the B tree and B+ tree by considering the following elements.

   d) 1,5,2,3,8,15,41,14

   e) 2,5,41,24,23,7,8,9,12

   f) 1,2,3,4,5,6

iii) Write the steps to insert an element in a B and B+ tree.

4.12 REFERENCES AND SUGGESTED READINGS

- Thareja, Reema. *Data structures using C*. Oxford University Press, Inc., 2011.
- Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.

Space for learners: