



Semester- II

MSc-IT
Paper: INF 2056

Advanced Data Structure

www.idolgu.in

GAUHATI UNIVERSITY
Institute of Distance and Open Learning

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

SECOND SEMESTER

(under CBCS)

M.Sc.- IT

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

ADVANCED DATA STRUCTURE



Contents:

BLOCK I: REVIEW OF BASIC CONCEPTS IN DATA STRUCTURE

- Unit 1 : Introduction to Data Structure
- Unit 2 : Linked List
- Unit 3 : Stack and Queue
- Unit 4 : Binary tree and Binary Search Tree

BLOCK II: DICTIONARY ADT AND SORTING AND SELECTION TECHNIQUES

- Unit 1 : Introduction to Search Trees
- Unit 2 : AVL trees and Red-Black trees
- Unit 3 : Multi way search trees, 2-3 trees and Splay trees
- Unit 4 : Hashing
- Unit 5 : Sorting Algorithms and Selection Techniques I
- Unit 6 : Sorting Algorithms and Selection Techniques II

BLOCK III: PRIORITY QUEUE ADT, PARTITION ADT AND DATA STRUCTURE FOR EXTERNAL STORAGE OPERATIONS

- Unit 1 : Priority Queue ADT I
- Unit 2 : Priority Queue ADT II
- Unit 3 : Partition ADT
- Unit 4 : B Tree and B+ Tree

Contributors:

Mr. Ajit Das Asstt. Prof, Dept. of Computer Science and Technology Bodoland University Kokrajhar(BTAD), Assam	(Block I : Units- 1 & 4) (Block II : Units- 4, 5 & 6)
Dr. Swapnanil Gogoi Asstt. Prof., GUIDOL	(Block I: Units- 2 & 3)
Mr. Manash K. Gogoi Asstt. Prof, Dept. of Computer Science Handique Girls' College, Guwahati, Assam	(Block II: Units- 1, 2 & 3)
Dr. Utpal Barman Asstt. Prof., Dept. of Computer Science and Engineering GIMT, Guwahati, Assam	(Block III: Units- 1, 2, 3 & 4)

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
-----------------------	--------------------------

ISBN:

August, 2021

© Copyright by IDOL, Gauhati University. All rights reserved. No part of this work may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise.
Published on behalf of Institute of Distance and Open Learning, Gauhati University by the Director, and printed at Gauhati University Press, Guwahati-781014.

Unit 1: Introduction to Data Structure

CONTENTS:

- 1.0 Introduction to Data Structures:
- 1.1 Unit objective
- 1.2 Classification of Data structures
 - 1.2.1 Linear Data structure
 - 1.2.2 Non-linear Data Structure
 - 1.2.3 Primitive data structures.
 - 1.2.4 Non-primitive data structures.
 - 1.2.5 Differences between primitive and non primitive data structures
- 1.3 What is an Algorithm?
 - 1.3.1 Space Complexity
 - 1.3.2 Time Complexity
- 1.4 Abstract Data Type(ADT)
- 1.5 What is an Array?
 - 1.5.1 Traverse
 - 1.5.2 Searching elements in an array
 - 1.5.2.1 Linear or sequential search
 - 1.5.2.2 Binary Search.
 - 1.5.3 Insertion operation of array
 - 1.5.4 Deletion in an array
- 1.6 Summing Up
- 1.7 Key terms
- 1.8 Questions and Answer
 - 1.8.1 Multiple choice Question and Answer
 - 1.8.2 Fill up the Blanks and Answer
 - 1.8.3 Short types question
 - 1.8.4 Broad type question
- 1.9 Suggested Readings

1.0 Introduction to Data Structures:

Since the beginning of computer invention, people are using the term data to describe the computer information. **Data** 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 become 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 we can represent data structure as : Algorithm + Data structure = Program.

Data structures and algorithms both are inter-related 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..

1.1. Unit Objective:

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

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

1.2. Classification of Data structures:

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 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.

<p>For example</p> <p>int, float, char, pointer etc.</p>	<p>For example</p> <p>Linked list, Tree, Graph, Stack, Queue etc.</p>
--	---

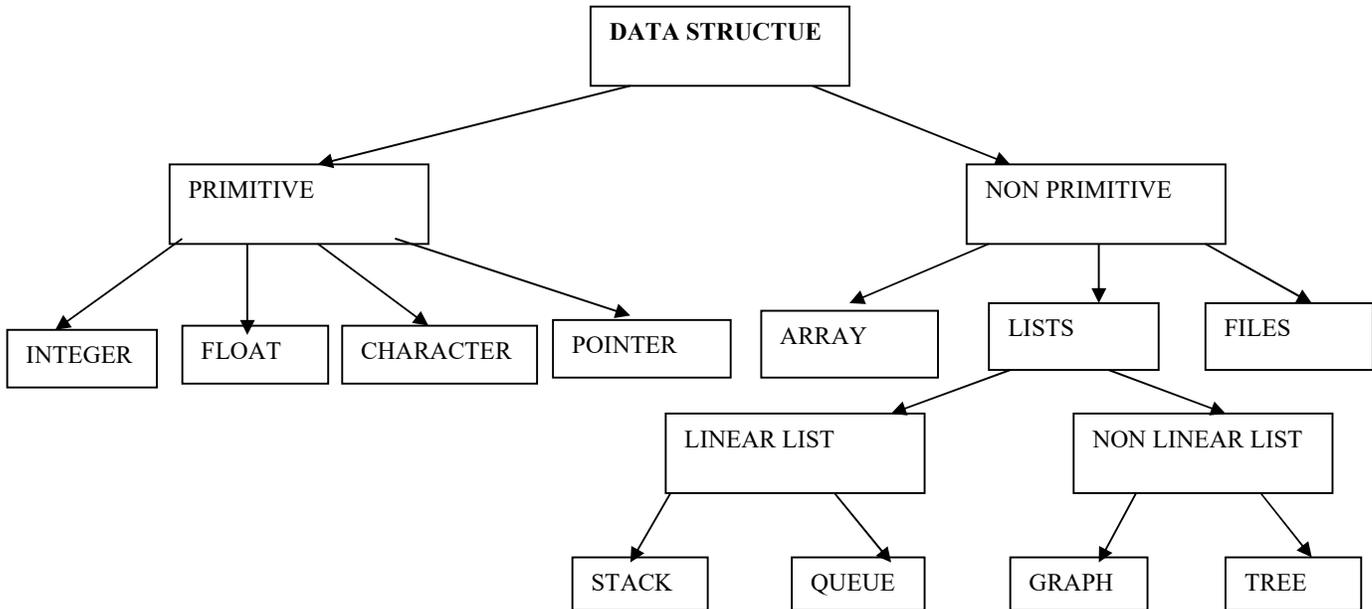


FIGURE 1: TYPES OF DATA STRUCTURE.

1.3 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 said to be efficient and fast. On the basis of following properties the performance of an algorithm is measured:

-Space Complexity

-Time Complexity

1.3.1 Space Complexity

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

An algorithm needed space for the following two components:

-A fixed part space is required to store certain data and variables which are not dependent of the size of the problem.

-A variable part space is required for variables, whose size is totally dependent on the size of the problem.

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.

1.3.2 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.

1.4 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.

1.5 What is an Array?

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 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 where it consists of 100 elements.

The subscript starts from zero, So age of zero elements is the first element of the array. Age of one element is the second element and so on.

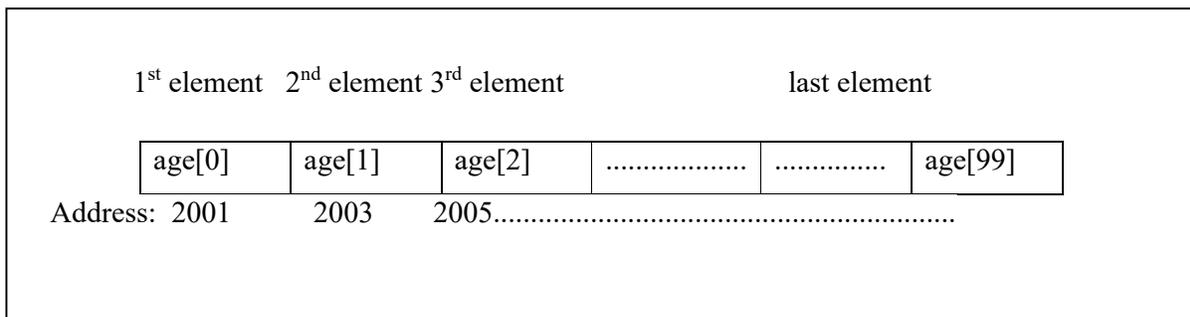


Figure 2: Structure of array elements

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.

1.5.1 Traverse – Visiting every element in the array exactly once.

Following C program traverses and prints the elements 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]);
}
```

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

1.5.2 Searching elements in an array:

Algorithm:

- a) Input the array elements.
- b) Input the element to be searched.
- c) Traverse the array first and check if the searched element is present in the array.
- d) Display "Yes" if the element present in the array, else display "No".

We can search elements by the following methods also:

1. Linear or sequential search
2. Binary search.

1.5.2.1 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 elements of the list one by one from the beginning.

Algorithm:

- 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

Array[2]= 7

Array[3]= 9

Array[4]= 3

Enter the searched item:

7

7 is found in the array

1.5.2.2 Binary Search.

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 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);
}

int bsearch(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

Array[3]= 9

Array[4]= 3

Enter the searched item:

6

6 is found at position 2 in the array

1.5.3 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.

Algorithm:

a is an array

n is number of elements (size)

Element is a data element

Pos is the location of the element to be inserted.

Insertion (a, n, Element, Pos)

Step 1: for i = n-1 downto Pos repeat step 2

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

End for

Step 3: $a[Pos] = Element$

Step 4: $n = n + 1$

Implementation of Algorithm through C language

```
#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");
    }

    if(pos<=0|| pos>n+1)
    {
        printf("Invalid position\n");
    }

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

```
    a[pos-1]=num;
    n++;
    printf("\n 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

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 7 8

1.5.4 Deletion in an array:

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

Algorithm:

a is an array

n is number of elements

Element is a deleted data element

Pos is the location of the element to be deleted.

Deletion (a, n, Pos)

Step 1: Element = a [Pos]

Step 2: for i = Pos to n-1 repeat step 3

Step 3: $a[i] = a[i+1]$

End for

Step 4: $n = n - 1$

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:

3

Updated array is

5 6 9

1.6 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 methods which can be implemented by using arrays-

- Linear or sequential search

- Binary search.

* In **linear searching method**, searching starts from the beginning of the list and continues till it finds the element or reaches the end of the list. The searched item is compared with each element 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 the item is found, the search will stop; otherwise, the array is divided into two halves: the first half contains all the elements to the left of the middle, and the other half contains all the elements to the right of the middle. Now all the elements on the left side will be smaller than the middle, and all the elements on the right side will be greater than the middle. Now if the searched element is less than the middle, it will search the elements in the left half; otherwise, it will search the elements in the right half. This process of comparing the elements with the middle element and dividing the array continues till the required item is found or a portion is reached that does not contain any elements.

* In **insertion in an array**, we will shift each and every element to the next position one by one from the insert element position.

* In **deletion of an element from an array**, we need to shift all the elements starting from the next element which is to be deleted one step towards the left.

1.7 Key Items:

- **Array**—An array data structure is a collection of data elements of the same type. Array elements are accessed by specifying an integer index starting from 0.

- **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. We can represent data structure as : Algorithm + Data structure = Program.
- **Pseudocode:** A Pseudocode is a technique to represent the implementation of an algorithm in such way that it should easily understand with basic knowledge of programming. It will describe the basic concept of a program without syntax of any programming languages.
- **Flowchart:** A flowchart is a diagrammatic representation of logic sequence that shows a step by step progress of solving a particular problem.
- **Element**—An item in an array.
- **Index**— Array element's position number. An index either must be zero, a positive integer, or an expression that evaluates to zero or a positive integer.
- **One-dimensional array**—An array that uses only one index. Conceptually you can think of a one-dimensional array as a row, where elements are stored one after another. Data stored here linearly.

1.8 Questions and Answers

1.8.1 Multiple Choice Questions:

1. From below which one is non-linear type of data structure?
 - a) Array
 - b) Stack
 - c) Queue
 - d) Graph
2. From below which one is linear type data structure?
 - a) Graph
 - b) Tree
 - c) AVL tree
 - d) Queue.
3. From below which one is primitive data structure?
 - a) Linked list,

- b) Tree
 - c) Pointer
 - d) Graph
4. From below which one is non primitive data structure?
- a) int
 - b) Stack
 - c) Float
 - d) Char
5. _____ 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
6. 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
7. 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
8. Which searching techniques not required elements to be sorted?
- a) Linear search
 - b) Binary search
 - c) Interpolation search
 - d) All of the above.

9. 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.

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

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

Solution:

1. From below which one is non-linear type of data structure?

Answer: d) Graph .

Explanation: Array, Queue and STACK all are linear data structure.

2. From below which one is linear type data structure?

Answer : c) Queue

Explanation: Graph, Tree and AVL tree all are non linear data structure. Only Queue is linear Data Structure. Because data store here in linear order.

3. From below which one is non-linear type of data structure?

Answer: d) Graph .

Explanation: Array, Queue and STACK all are linear data structure.

4. From below which one is primitive data structure?

Answer: c) Pointer.

5. From below which one is non primitive data structure?

Answer: b) Stack

6. _____ 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.

Answer: b) Abstract Data type

7. What are the advantages of arrays?

Answer:

d

Explanation: Arrays store Same type of Data and present in continuous memory locations.

8. What are the disadvantages of arrays?

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

Explanation: Arrays are of fixed size. If we insert elements less than the allocated size, unoccupied positions can't be used again. Wastage will occur in memory.

9. Which searching techniques not required elements to be sorted?

Answer: a) Linear Search

10. In _____ we will shift each and every element to the next position one by one from location.

Answer a) Insertion in array

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

Answer: b) Deleted.

1.8.2 Fill up the blanks:

1. Integers, character constants, floating point numbers, string constants and pointers are _____ type of Data Structure.

2. Linked list, Tree, Graph, Stack, Queue are _____ type of Data Structure.

3. Space complexity is function represents the amount of _____ space needed the algorithm for implementation.
4. Time Complexity of an algorithm is a function representing the amount of _____ required by the algorithm to be executed.
5. Array elements are stored in _____ memory cells.
6. In binary search the elements in the array need to be _____ .
7. 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 _____ .

Answer:

1. Primitive
2. Non Primitive
3. Memory
4. time
5. Consecutive
6. sorted.
7. left

1.8.3 Short Answer type Question:

1. What is Data Structure?
2. What do you mean by an Algorithm
3. What is Pseudocode?
4. What is Flowchart?

5. Explain Primitive Data Structure
6. Explain Non-Primitive Data Structure.
7. What is space complexity?
8. What is Time complexity?
9. What is Abstract Data type?
10. What is an Array?

1.8.4 Long Type Question:

2. Explain the differences between Primitive and No-primitive data structure.
3. Write a C program to implement the Linear Search Algorithm.
4. Write the algorithm of Binary Search.
5. Explain the procedure of insertion an element in any position in an array.
6. Explain the procedure of deletion an element in any position from an array.

1.9 Suggested Readings:

1. **Data Structures Through C In Depth.** by S.K.Srivastava /Deepali Srivastava..
2. **Data Structures Using C** by Reema Thareja

UNIT 2: LINKED LIST

UNIT STRUCTURE

- 2.1 Introduction
- 2.2 Objectives
- 2.3 Basics of Linked List
 - 2.3.1 Types of Linked List
 - 2.3.2 Comparison of Linked List and Array
 - 2.3.3 Applications of Linked List
- 2.4 Singly Linked List
 - 2.4.1 Insert Operation on Singly Linked List
 - 2.4.2 Delete Operation on Singly Linked List
 - 2.4.3 Traversal Operation on Singly Linked List
- 2.5 Doubly Linked List
 - 2.5.1 Insert Operation on Doubly Linked List
 - 2.5.2 Delete Operation on Doubly Linked List
 - 2.5.3 Traversal Operation on Doubly Linked List
- 2.6 Circular Linked List
 - 2.6.1 Insert Operation on Circular Linked List
 - 2.6.2 Delete Operation on Circular Linked List
 - 2.6.3 Traversal Operation on Circular Linked List
- 2.7 Doubly Circular Linked List
- 2.8 Summing Up
- 2.9 Possible Questions
- 2.10 References and Suggested Readings

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 OBJECTIVE

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.

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.

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 memory address of the next data for data traversal operation.	Data may not be stored in contiguous memory locations in linked list. So each node in a 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.

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

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, 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'.

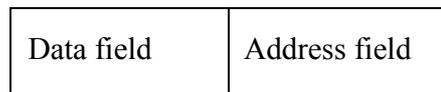


Figure 2.1: Diagrammatic Representation of a Singly Linked List Node

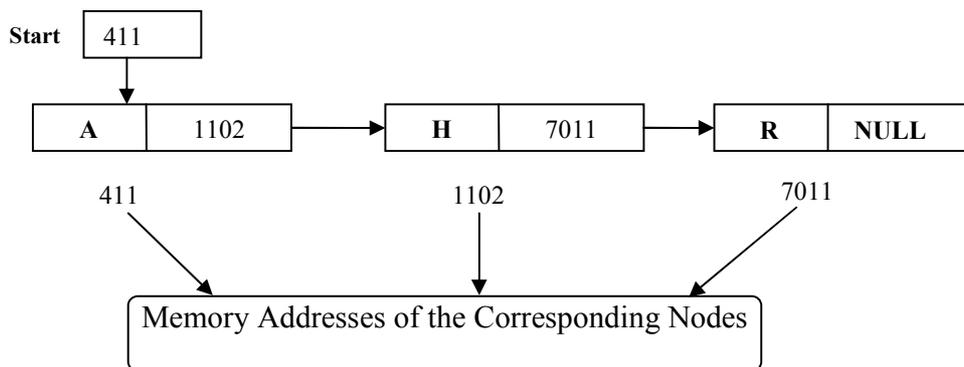


Figure 2.2: Diagrammatic Representation of a Singly Linked List

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.

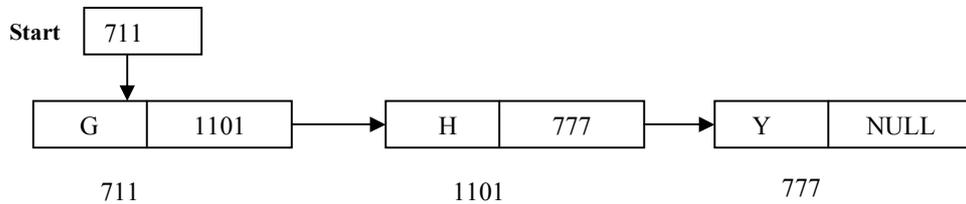


Figure 2.3: A Singly Linked List With Three Existing Nodes

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.

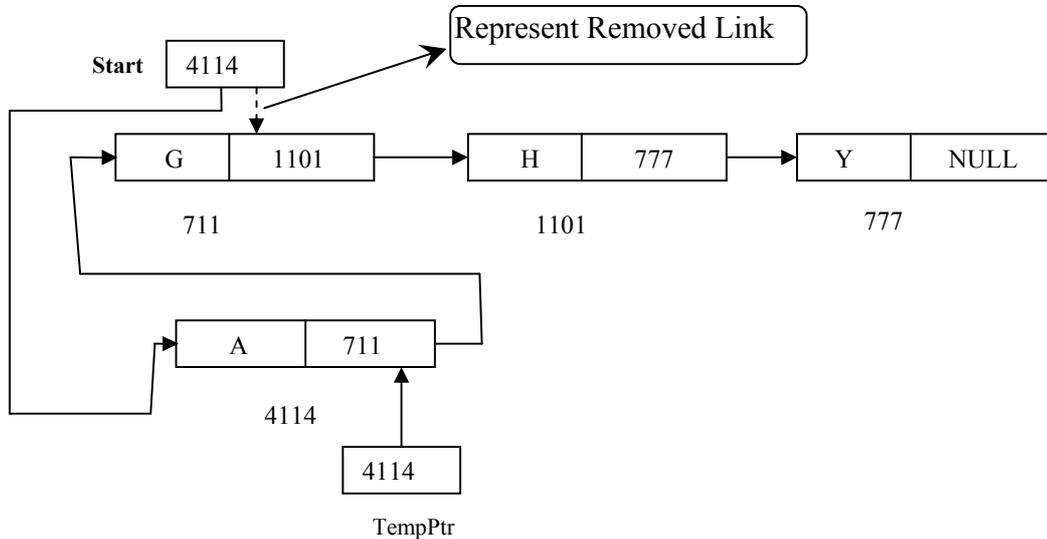


Figure 2.4: Singly Linked List After Insertion Of A New Node At First Position

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.

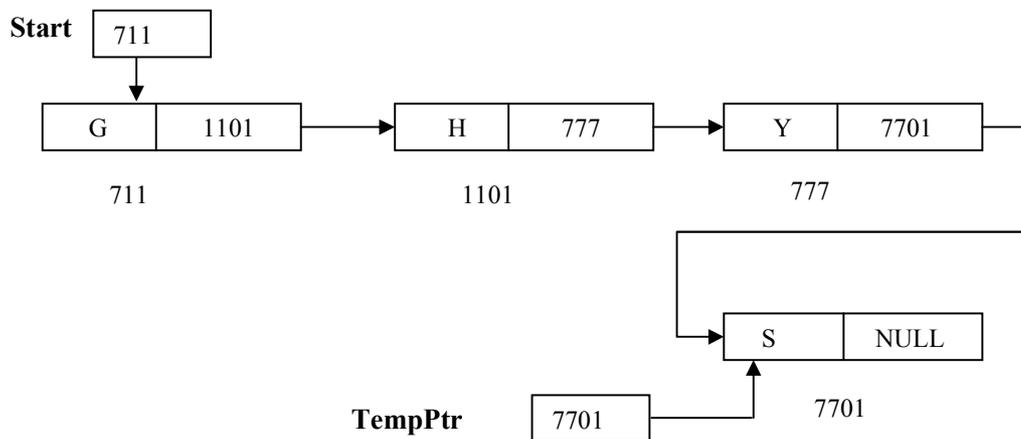


Figure 2.5: Singly Linked List After Insertion Of A New Node At Last Position

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 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.

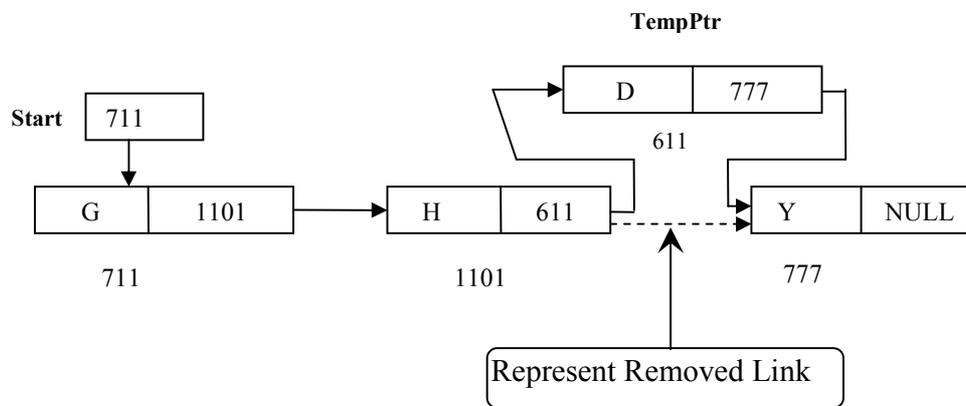


Figure 2.6: Singly Linked List After Insertion Of A New Node At 3rd Position

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;

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;
        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;
        }
    }
}
```

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

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.

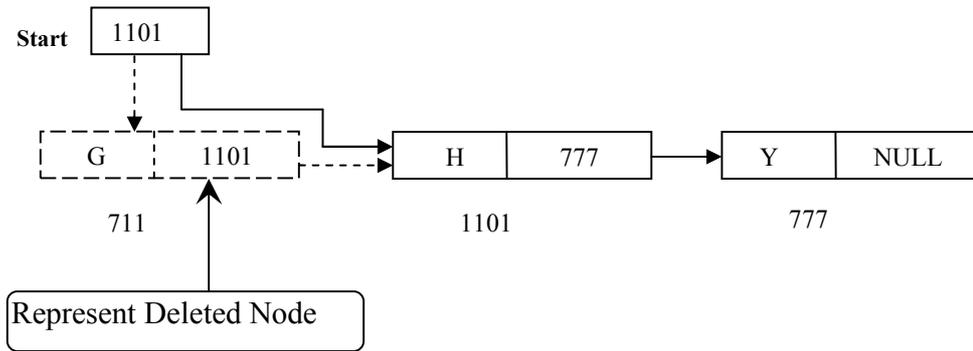


Figure 2.7: Singly Linked List After Deletion Of The First Node

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 nodes 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.

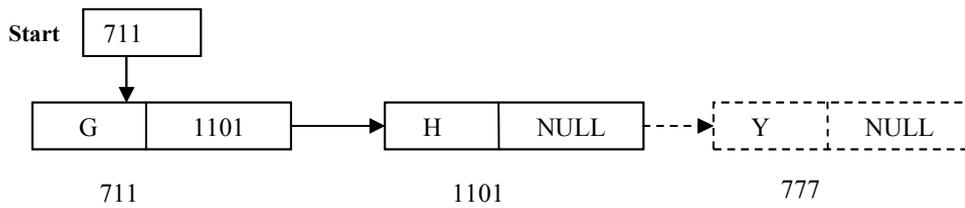


Figure 2.8: Singly Linked List After Deletion Of The Last Node

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.

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

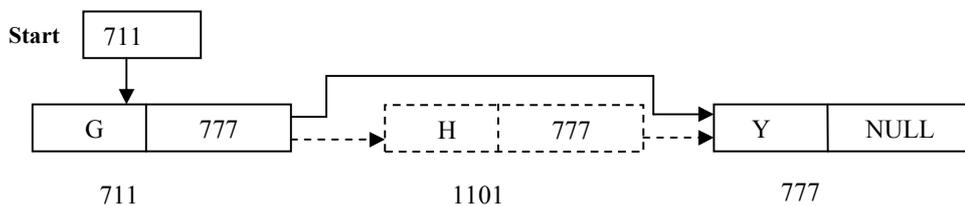


Figure 2.9: Singly Linked List After Deletion Of The 2nd Node

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();
};

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;
    }
}

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
    {
        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
                {
                    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;  
}
```

2.4.3 Traversal 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.

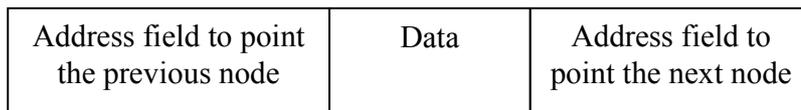


Figure 2.10: Diagrammatic Representation of a Doubly Linked List Node

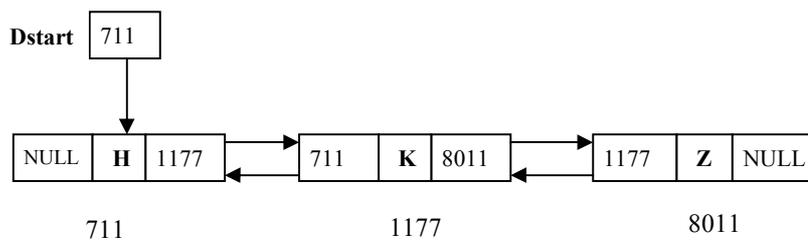


Figure 2.11: Diagrammatic Representation of a Doubly Linked List

<p style="text-align: center;">STOP TO CONSIDER</p> <p style="text-align: center;">Any node can be reached from any node in a Doubly linked list.</p>
--

2.5.1 Insert Operation on Doubly Linked List

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.

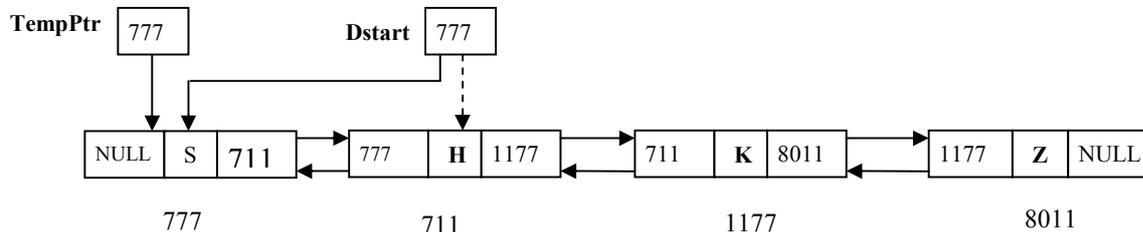


Figure 2.12: Doubly Linked List After Insertion Of A New Node At The First Position

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.

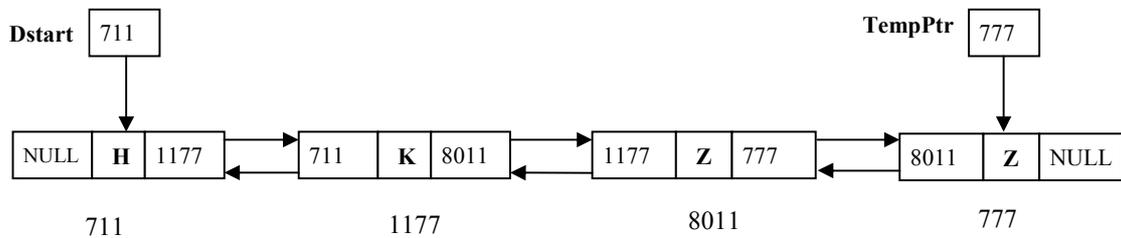


Figure 2.13: Doubly Linked List After Insertion Of A New Node At The Last Position

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 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.

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.

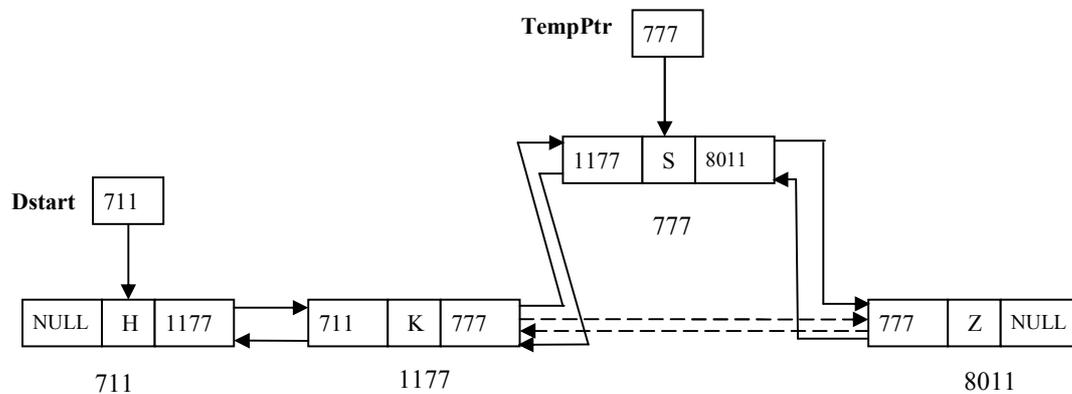


Figure 2.14: Doubly Linked List After Insertion Of A New Node At The 3rd Position

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.
- 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.

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;
    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;
    }
    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";
    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;
}
```

2.5.2 Delete Operation on Doubly Linked List

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.

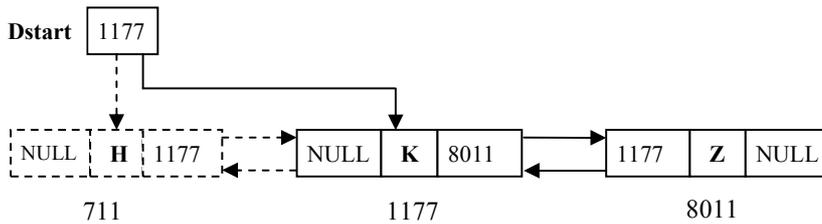


Figure 2.15: Doubly Linked List After Deletion Of The First Node

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.

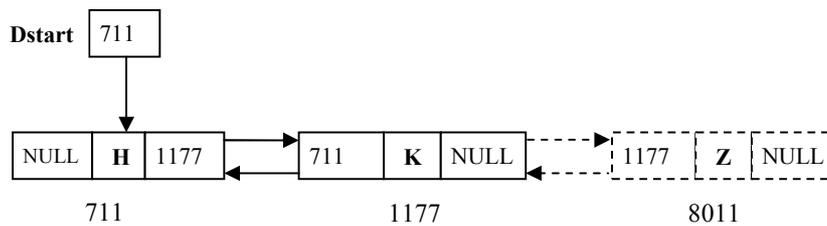


Figure 2.16: DoublyLinked List After DeletionOf The Last Node

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.

- 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.

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

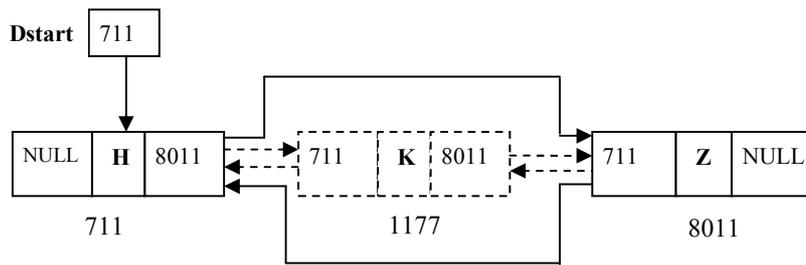


Figure 2.17: DoublyLinked List After DeletionOf The 2nd Node

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;
            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
    {
        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=";\n cin>>S_Pos;\n if(Dstart==NULL)\n {\n cout<<"\n Empty linked list";\n return(0);\n }\n else\n {\n DeletePtr=Dstart;\n if(S_Pos==1)\n {\n if(Dstart->next==NULL)\n Dstart=NULL;\n else\n {\n Dstart=Dstart->next;\n Dstart->prev=NULL;\n }\n delete DeletePtr;\n return(S_Pos);\n }\n else\n {\n while(DeletePtr->next!=NULL && count<S_Pos)\n {\n count=count+1;\n PtrPrev=DeletePtr;\n DeletePtr=DeletePtr->next;\n }\n if(count==S_Pos)\n {\n PtrPrev->next=DeletePtr->next;\n if(DeletePtr->next!=NULL)\n (DeletePtr->next)->prev=PtrPrev;\n delete DeletePtr;\n return(S_Pos);\n }\n 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;
            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();
                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;
}
```

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 contain 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.

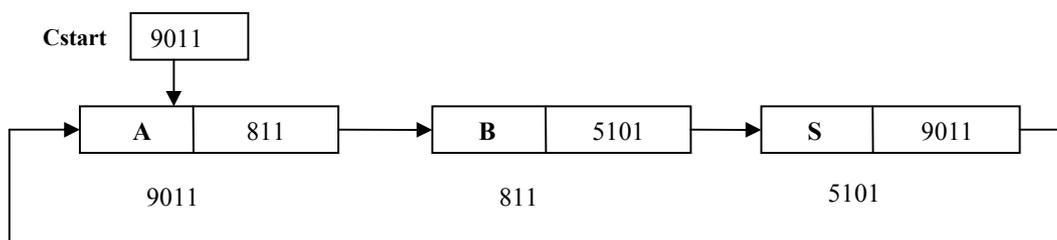


Figure 2.18: Diagrammatic Representation of a Circular Linked List.

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.

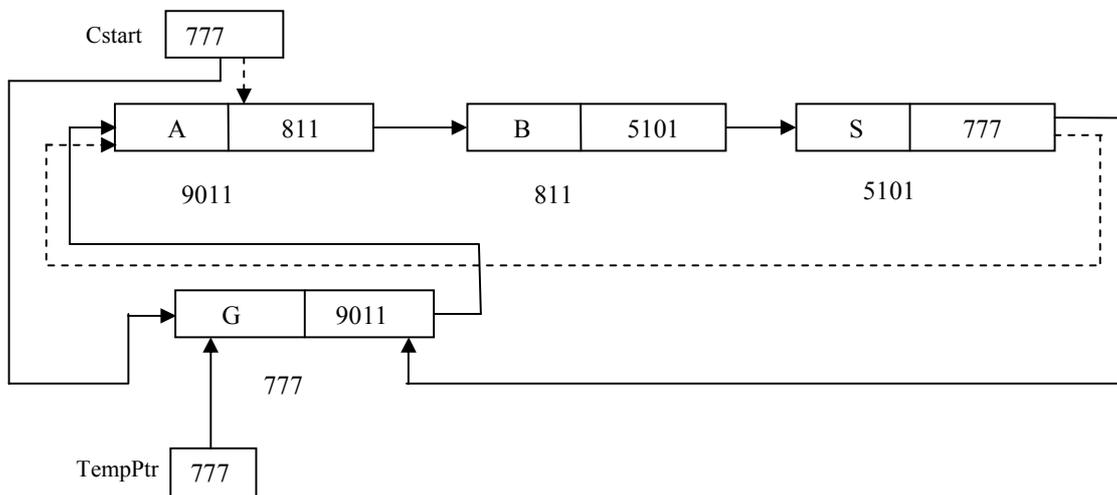


Figure 2.19: CircularLinked List After Insertion Of A New Node At The First Position

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.

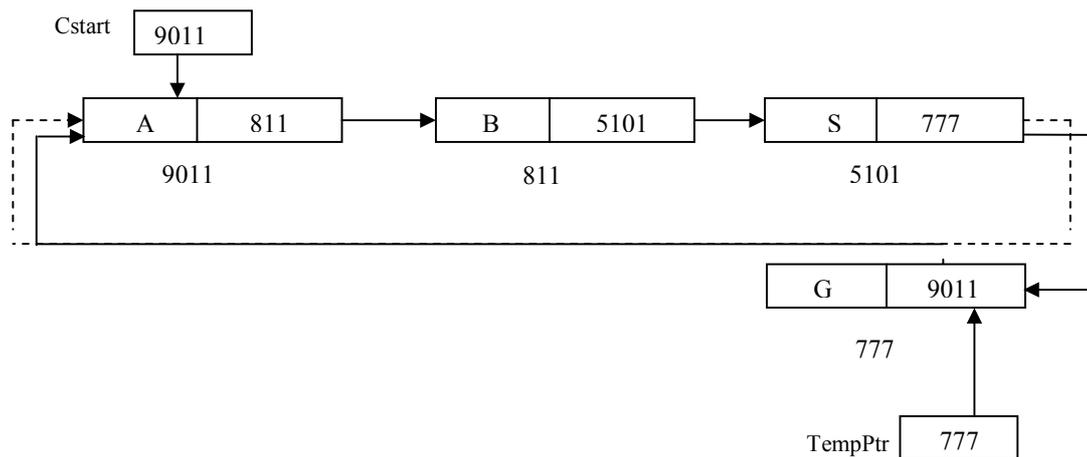


Figure 2.20: Circular Linked List After Insertion Of A New Node At The Last Position

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.

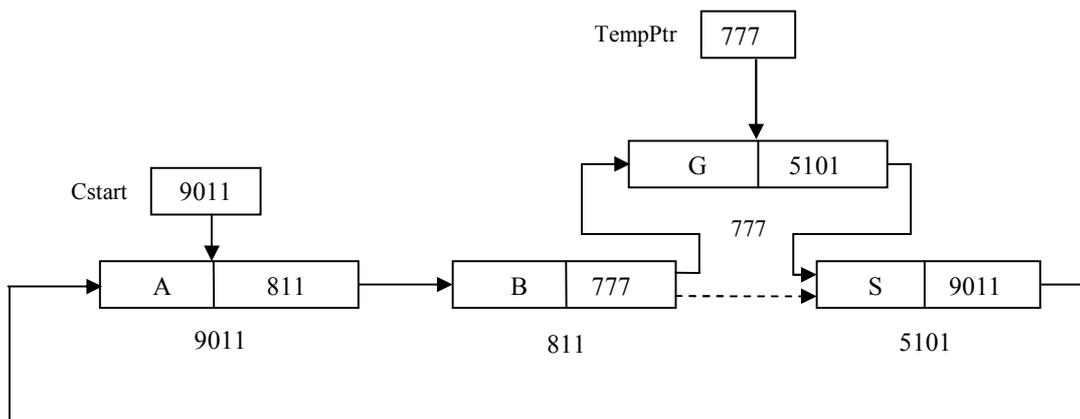


Figure 2.21: CircularLinked List After Insertion Of A New Node At The 3rd Position

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
{
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;
    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
    {
        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";
    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
            {
                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;
}
```

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.

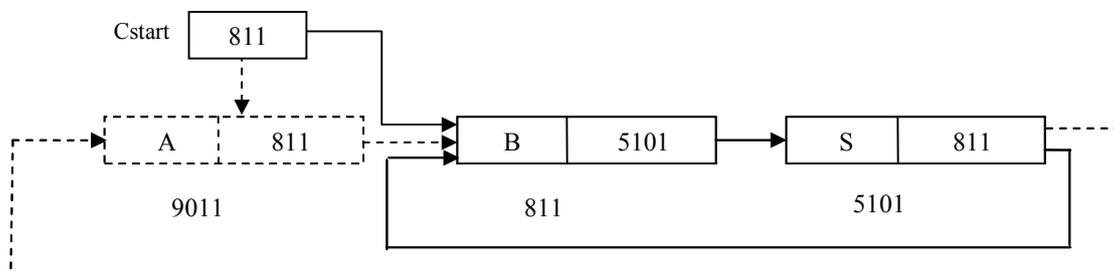


Figure 2.22: CircularLinked List After DeletionOf The First Node

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.

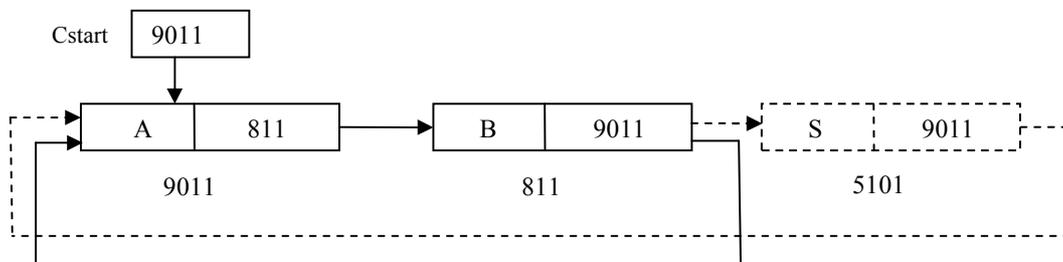


Figure 2.23: Circular Linked List After Deletion Of The Last Node

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.

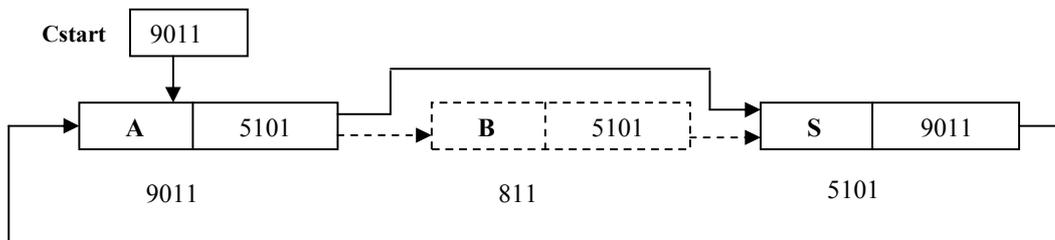


Figure 2.24: Circular Linked List After Deletion Of The 2nd Node

2.6.3 Traversal Operation on Circular Linked List

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;

    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;  
                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);
}
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;
        }
        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";
}

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

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 on the Doubly circular linked list that is presented in figure 2.25. These operations in Doubly circular linked list are implemented in Program 2.7.

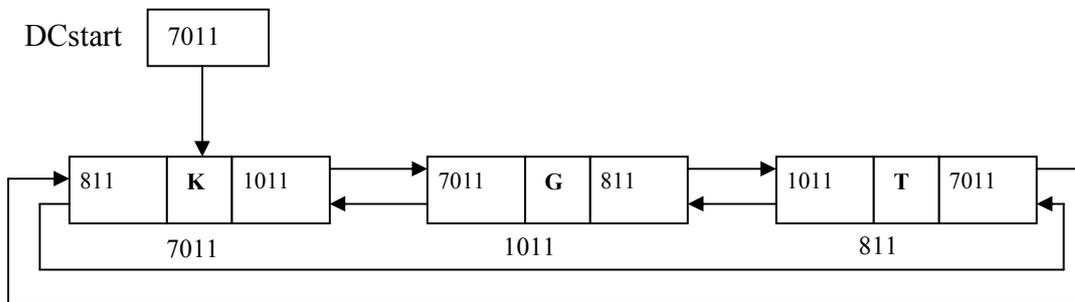


Figure 2.25: Diagrammatic Representation of a Doubly Circular Linked List.

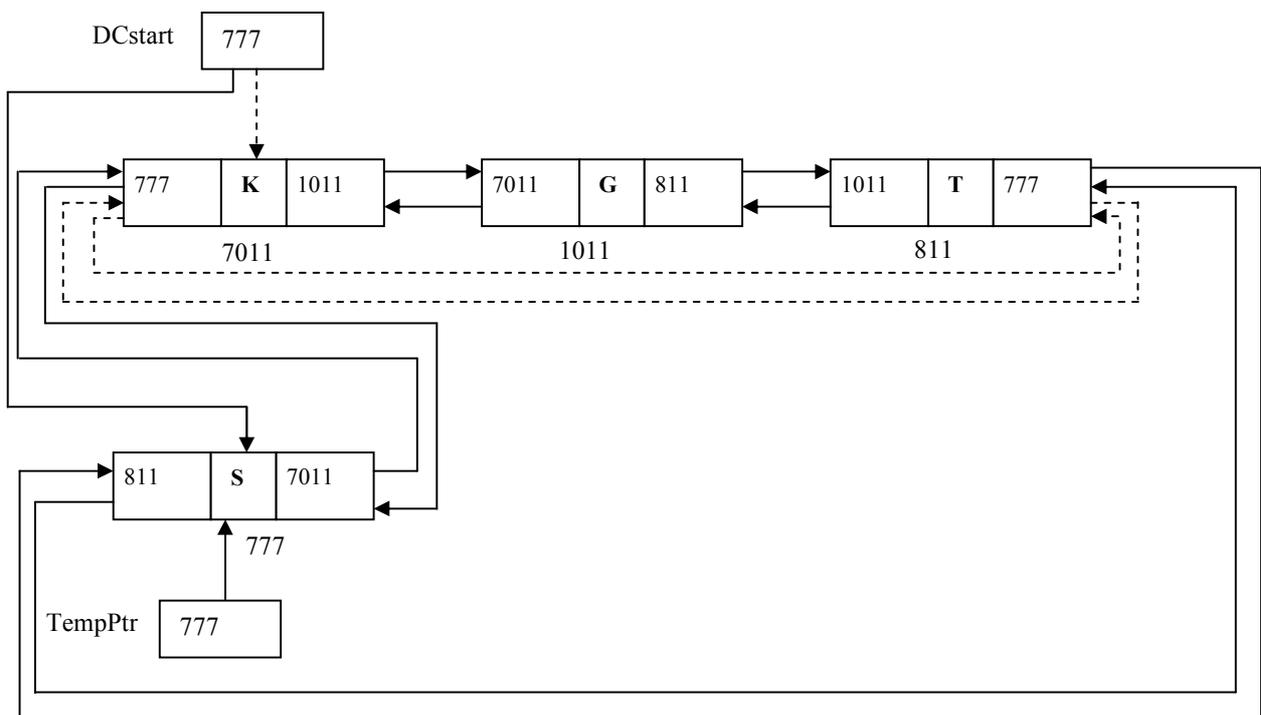


Figure 2.26: Doubly Circular Linked List After Insertion of a Node at First Position

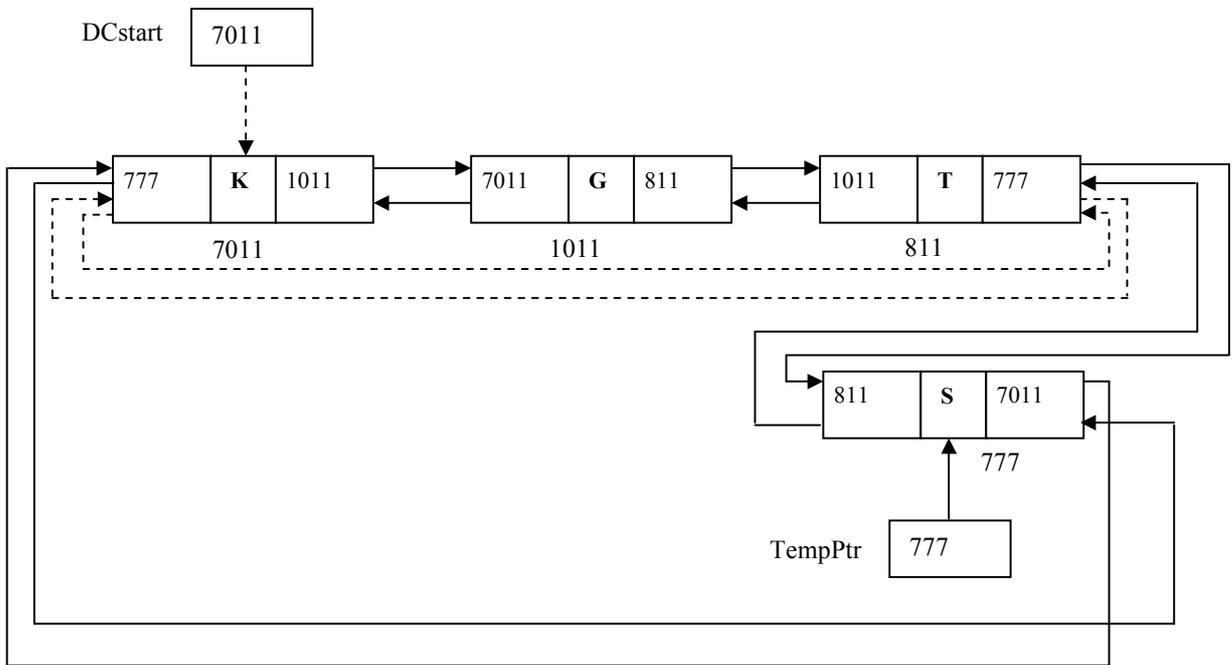


Figure 2.27: Doubly Circular Linked List After Insertion of a Node at Last Position

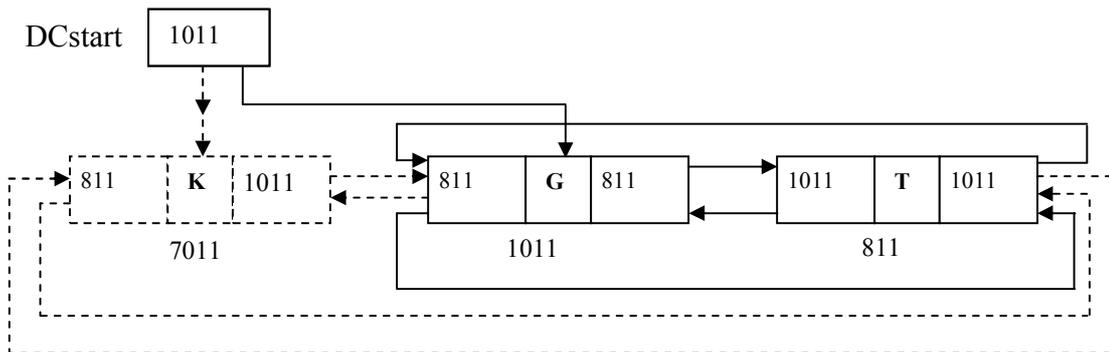


Figure 2.28: Doubly Circular Linked List After Deletion of the First Node

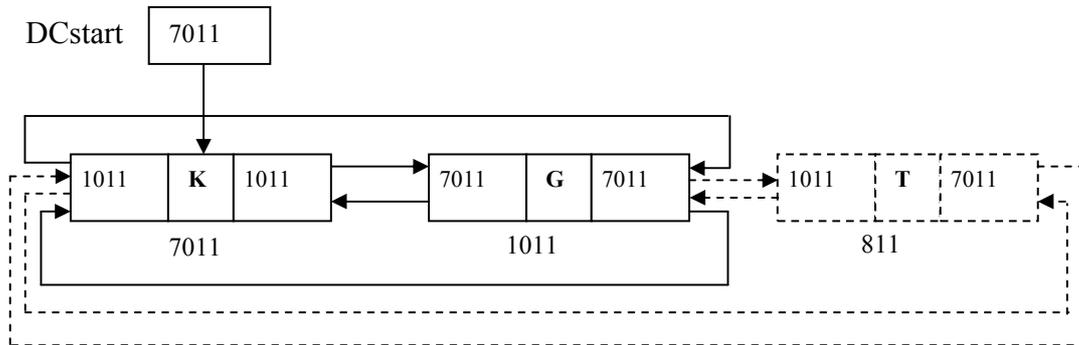


Figure 2.29: Doubly Circular Linked List After Deletion of the Last Node

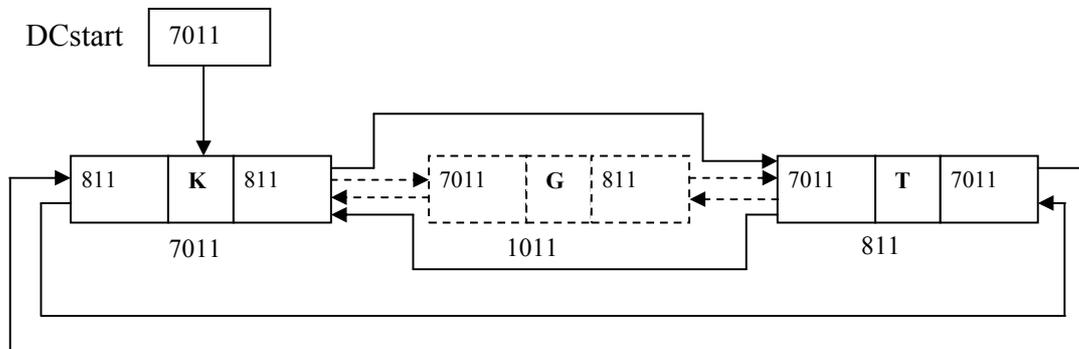


Figure 2.30: Doubly Circular Linked List After Deletion of the node available at 2nd Position.

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()
        {
            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++;
    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;
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)
    {
        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);
    }
}
}
```

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
    {
        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)  
        {  
            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";
    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;
    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::";
    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
- 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.

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.

ANSWER TO CHECK YOUR PROGRESS

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

2.9 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 5th node in a Circular linked list.
- 4) Write down a C++ program to insert a new node after 3rd 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?

2.10 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

UNIT-4: Binary Tree and Binary search Tree

CONTENTS:

- 4.0 Introduction
- 4.1 Unit objective
- 4.2 Definition of Tree
- 4.3 Some basic Terminology of tree:
 - 4.3.1 Node
 - 4.3.2 Root
 - 4.3.3 Edge
 - 4.3.4 Parent Node
 - 4.3.5 Child Node
 - 4.3.6 Siblings
 - 4.3.7 Leaf Node
 - 4.3.8 Internal Nodes
 - 4.3.9 Degree
 - 4.3.10 Level
 - 4.3.11 Height
 - 4.3.12 Path
 - 4.3.13 Sub Tree
- 4.4 Binary Tree:
 - 4.4.1 Types of Binary Tree
 - 4.4.1.1 Strictly binary Tree
 - 4.4.1.2 Full Binary tree
 - 4.4.1.3 Complete Binary Tree
 - 4.4.2 Representation of Binary Tree
 - 4.4.2.1 Array Representation of Binary Tree
 - 4.4.2.2 Linked list representation of Binary Tree
 - 4.4.2.3
 - 4.4.3 Traversal in binary Tree
 - 4.4.3.1 In-order Traversal (Root-Left-Right)
 - 4.4.3.2 Pre-order Traversal(Root-Left-Right)
 - 4.4.3.3 Post-order Traversal (Left-Right-Root)
- 4.5 Binary Search Tree

- 4.5.1 Traversal in Binary search tree
- 4.5.2 Searching in Binary search Tree
- 4.5.3 Creating a Binary Search tree
- 4.5.4 Insertion in Binary Search Tree
- 4.5.5 Delete an element from the Binary Search tree:
 - 4.5.5.1 The node to be deleted has no child i.e leaf node
 - 4.5.5.2 The node to be deleted has exactly one child
 - 4.5.5.3 The node to be deleted has exactly two children.

4.6 Summing Up

4.7 Questions and Answer

- 4.7.1 Multiple choice Question and Answer
- 4.7.2 Fill up the Blanks and Answer
- 4.7.3 Short types question
- 4.7.4 Broad type question

4.8 Suggested Readings

4.0 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.

4.1 Unit objective:

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.

4.2 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 $n \geq 0$ disjoint sets T_1, T_2, \dots, T_n , where each of this sets is also a tree. The sets T_1, T_2, \dots, T_n Are the subtrees of the root.

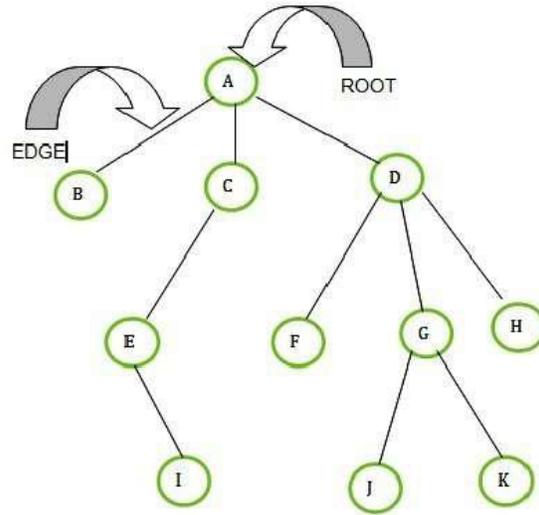
4.3 Some basic Terminology of tree:

4.3.1 Node:

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

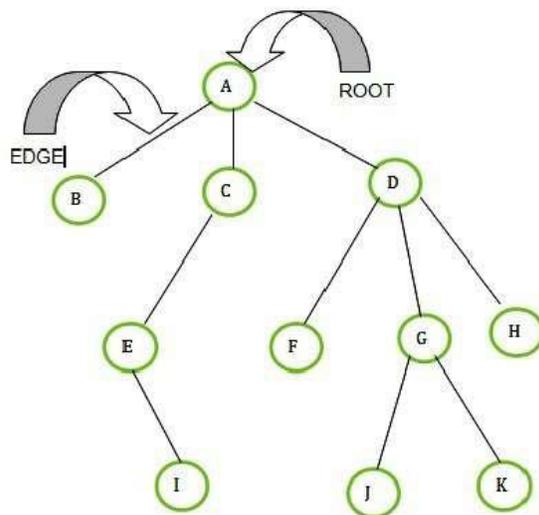
4.3.2 Root

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.



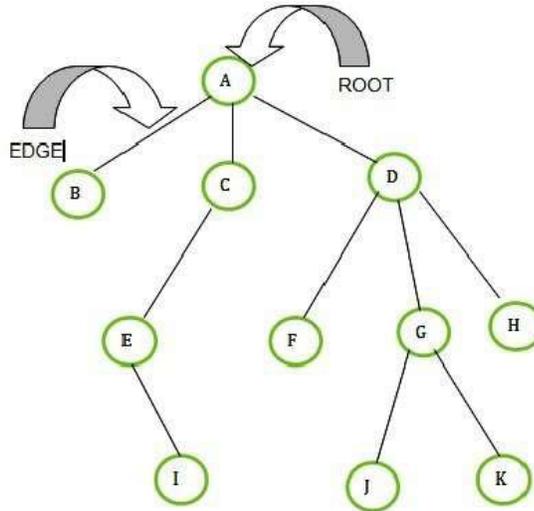
4.3.3 Edge

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.



4.3.4 Parent Node

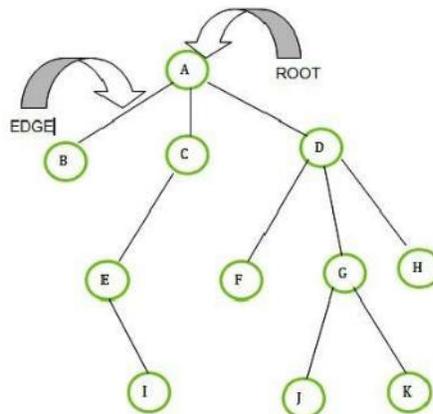
In a tree data structure, the immediate predecessor of a node is called as PARENT NODE. In simple words, the node which has a branch from it to any other node is called a parent node.



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

4.3.5 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

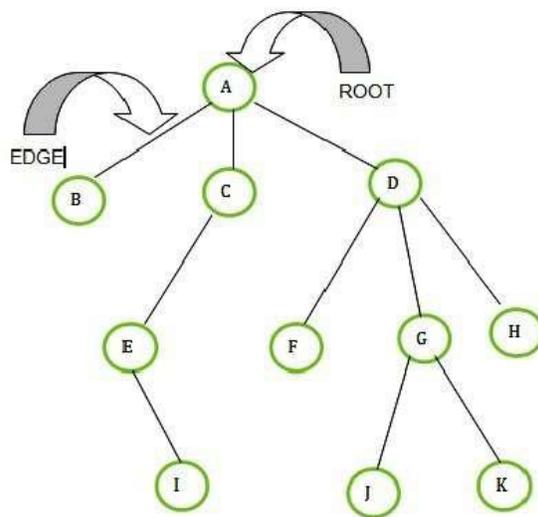
I is the children of E

F,G,H are children of D

J,K are children of G

4.3.6 Siblings

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



Here B,C,D are Siblings

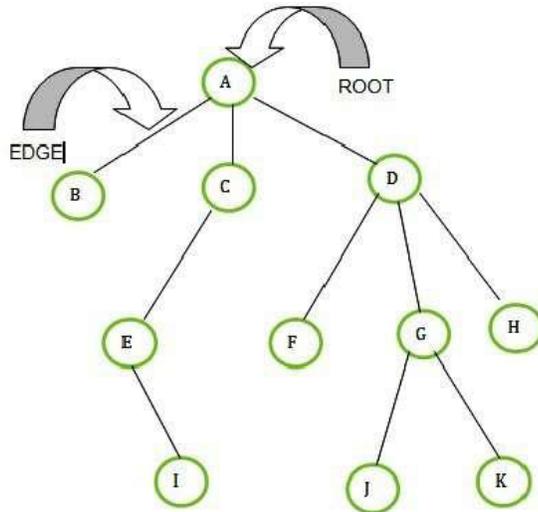
F,G,H are Siblings

J,K are Siblings

4.3.7 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.

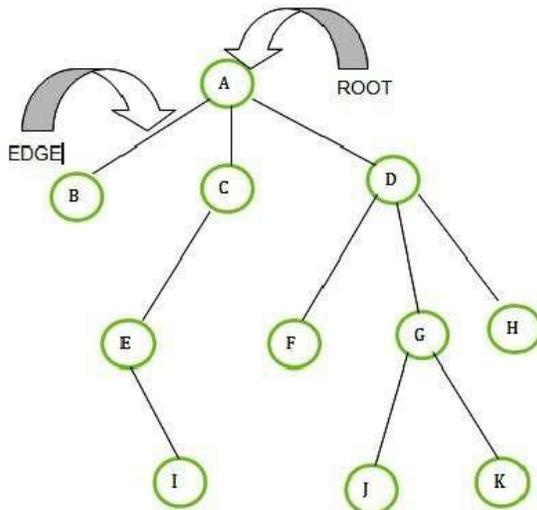
In a tree data structure, the leaf nodes are also called as External Nodes or 'Terminal' node.



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

4.3.8 Internal Nodes

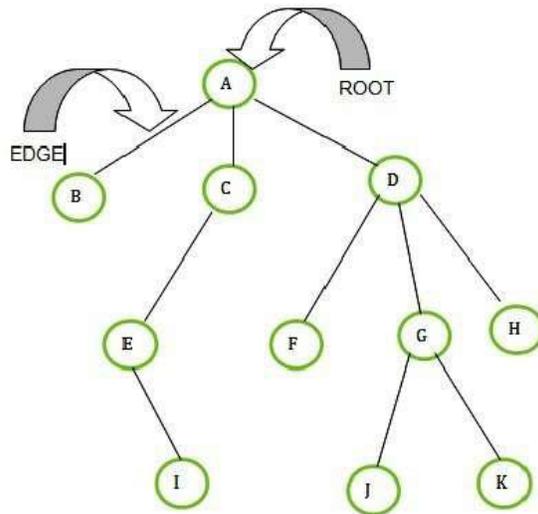
The node which has at least one child is called as INTERNAL Node. In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node It is also called as 'Non-Terminal' nodes.



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

4.3.9 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 subtrees or children is also called its Degree.



Here Degree of A is 3

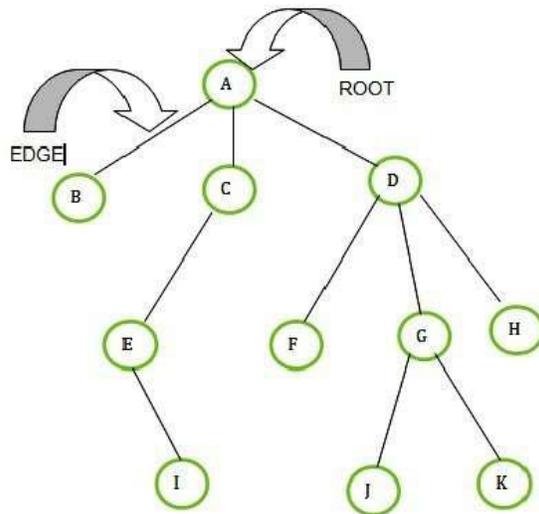
Degree of C is 1

Degree of B is 0

Etc.

4.3.10 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

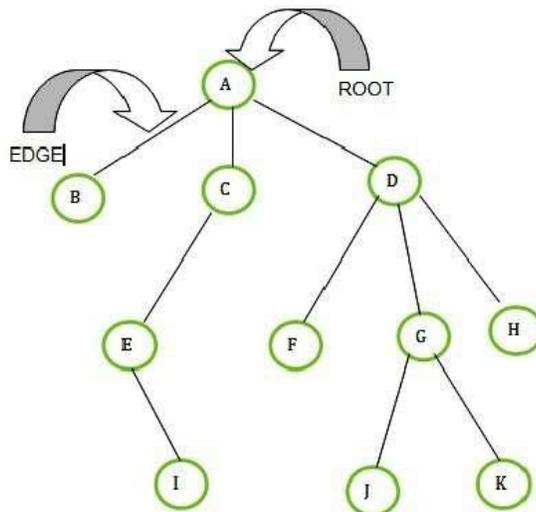
Level of B,C,D is 1

Level of E,F,G,H is 2

Level of I,J,K is 3

4.3.11 Height

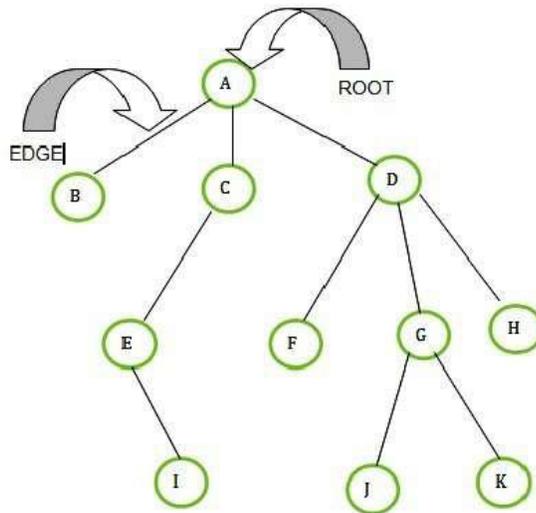
In a tree data structure, the total number of level in a tree is the height of the tree. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



Here Height of K is 0
Height of H is 1 etc.
Height of the tree is 3.

4.3.12 Path:

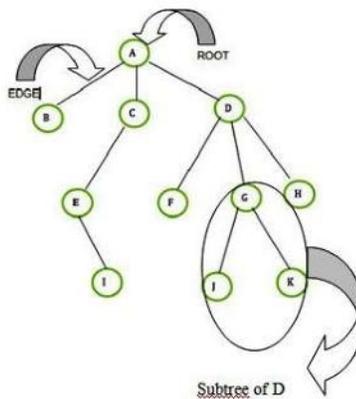
In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between the two Nodes.



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

4.3.13 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.

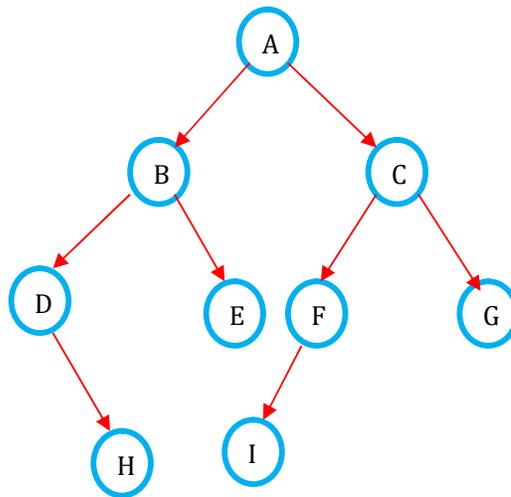


4.4 Binary Tree:

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.

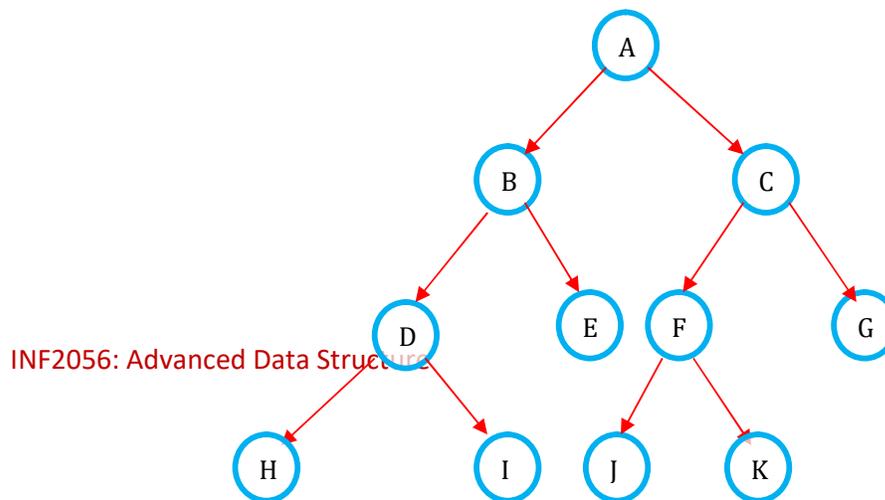


4.4.1 Types of Binary Tree:

There are different types of binary trees and they are:

4.4.1.1 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.



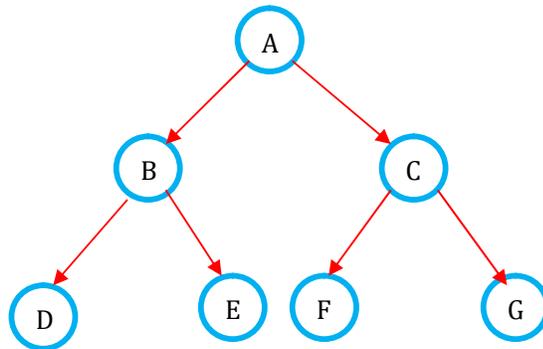
Property:

- 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.

4.4.1.2 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.

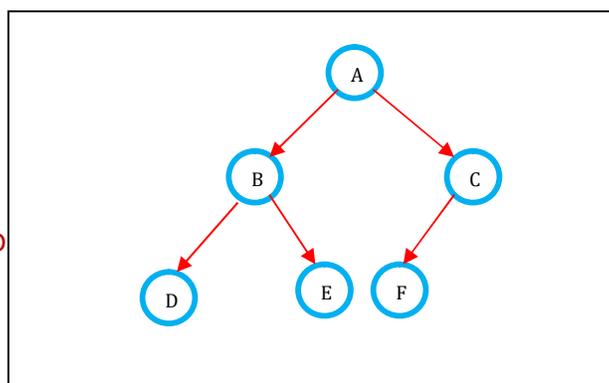
Here IF the number of any node is k , then the number of its left child is $2k$, the number of its right child is $2k+1$ and the number of its parent is $\text{floor}(k/2)$.

**4.4.1.3 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.

4.4.2 Representation of Binary Tree:

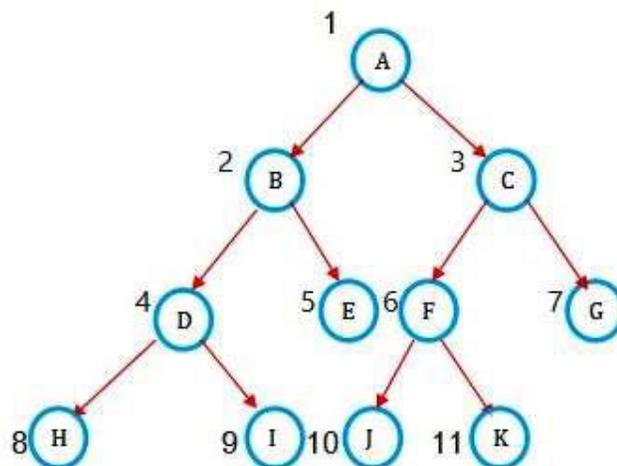
A binary tree data structure can be implemented by using two methods. Those methods are as follows...

-Array Representation

- Linked List Representation

4.4.2.1 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 give 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 –

$$\text{child1} = 2 * \text{parent}$$

$$\text{child2} = (2 * \text{parent}) + 1$$

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

$$\text{parent} = [\text{child} / 2]$$

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 consider 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.

4.4.2.2 Linked list representation of Binary Tree:

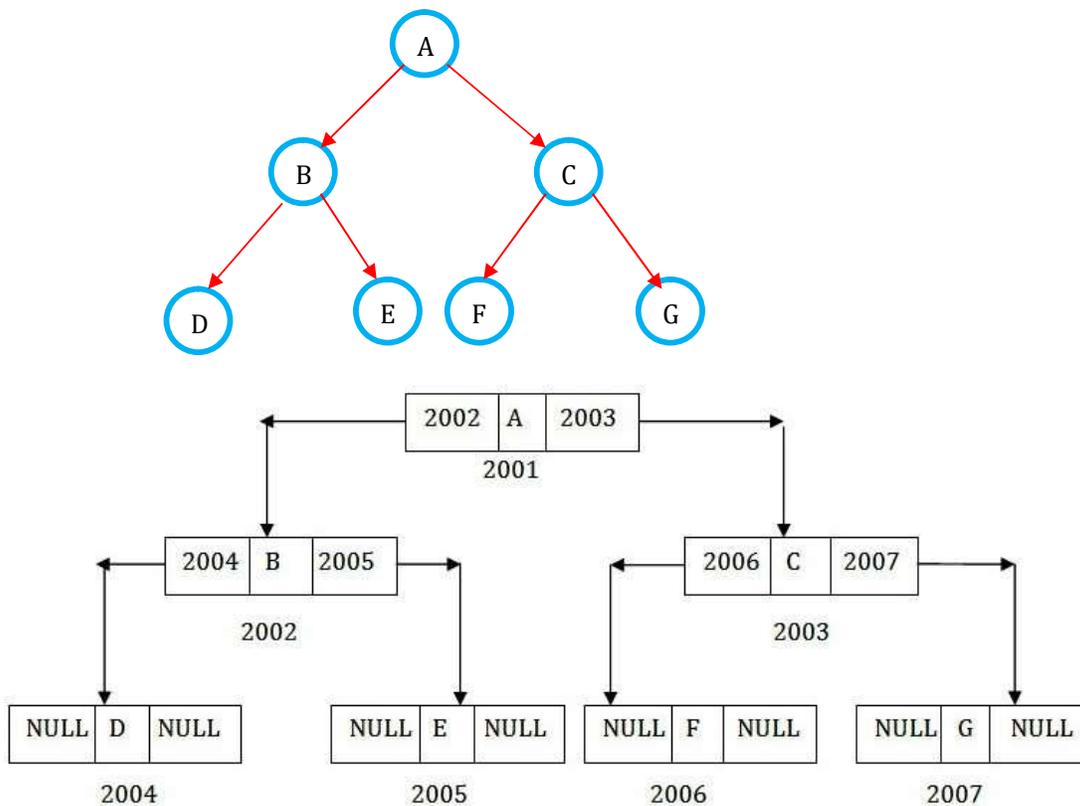
Already we know that linked list representation is better than array representation from memory point of view because in this representation explicit pointers are used to link the nodes of the tree. Here, we will use a double linked list to represent a binary tree. In a double linked list, every node consists of a data part and two link parts. Left link part store 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:



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.

4.4.3 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 always we start from the root node..
There are three ways which we use to traverse a tree –

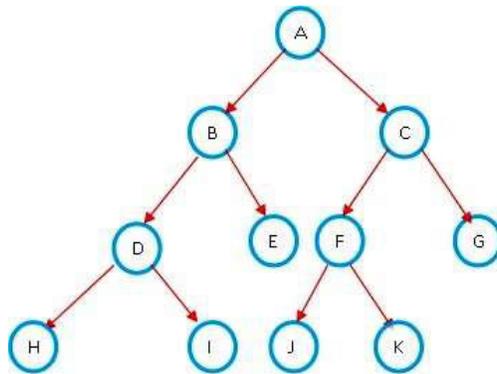
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

4.4.3.1 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)



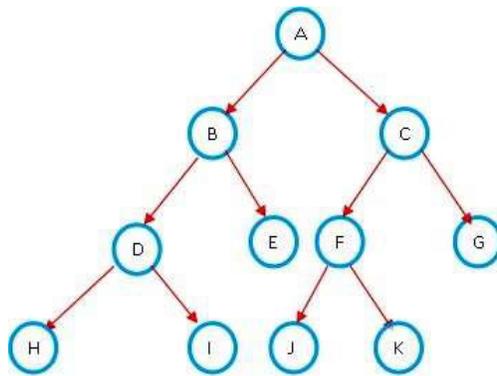
In-order traversal of the above mentioned tree is :
H-D-I-B-E-A-J-F-K-C-G

4.4.3.2 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 :

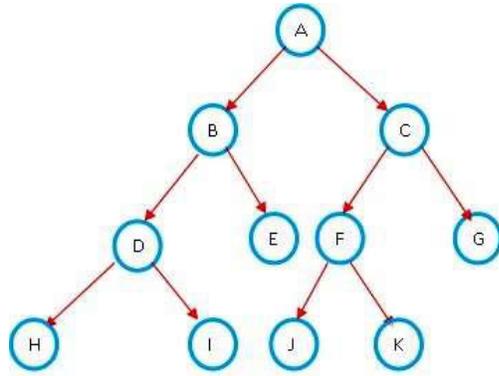
A-B-D-H-I-E-C-F-J-K-G

4.4.3.3 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 Postorder 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

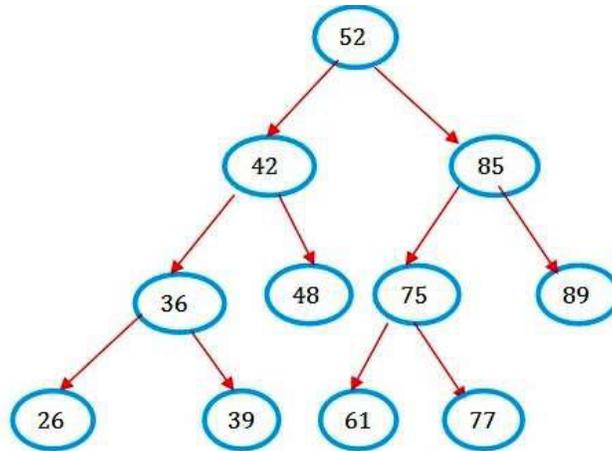
4.5 Binary Search Tree:

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.
- 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:



4.5.1 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

4.5.2 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 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
```

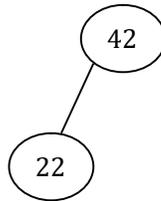
4.5.3 Creating a Binary Search tree:

Creating a Binary Search Tree from the key given below:

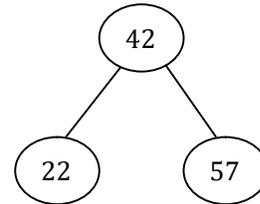
42, 22,57, 32,78,51,13, 54,82,69,29



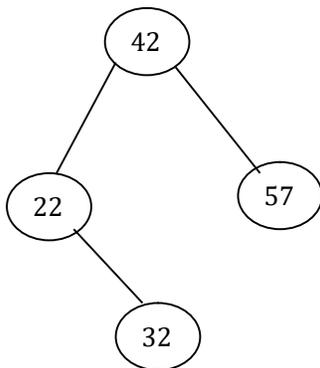
Insert 42



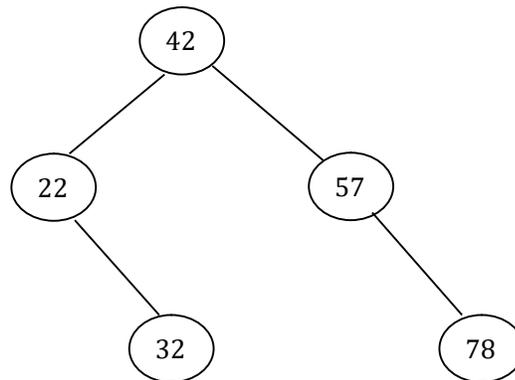
Insert 22



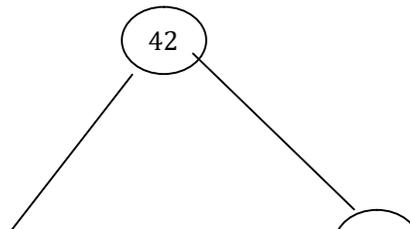
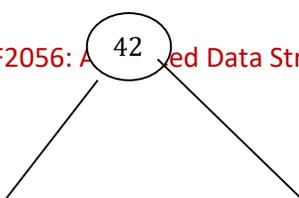
Insert 57



Insert 32



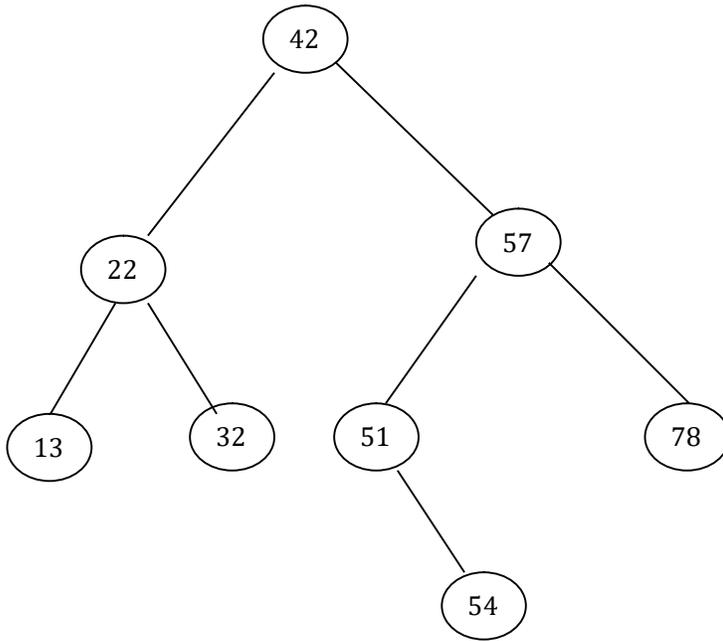
Insert 78



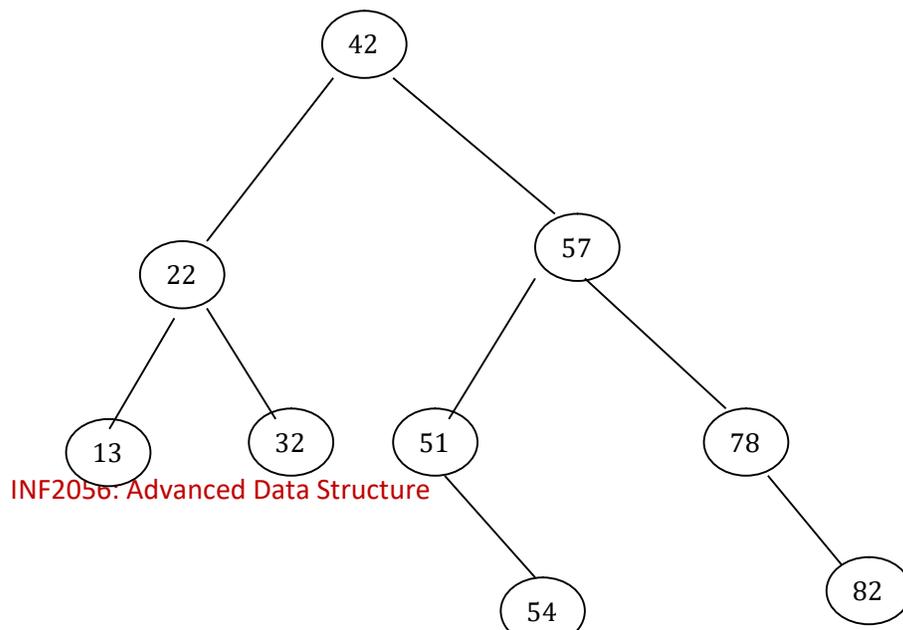


Insert 51

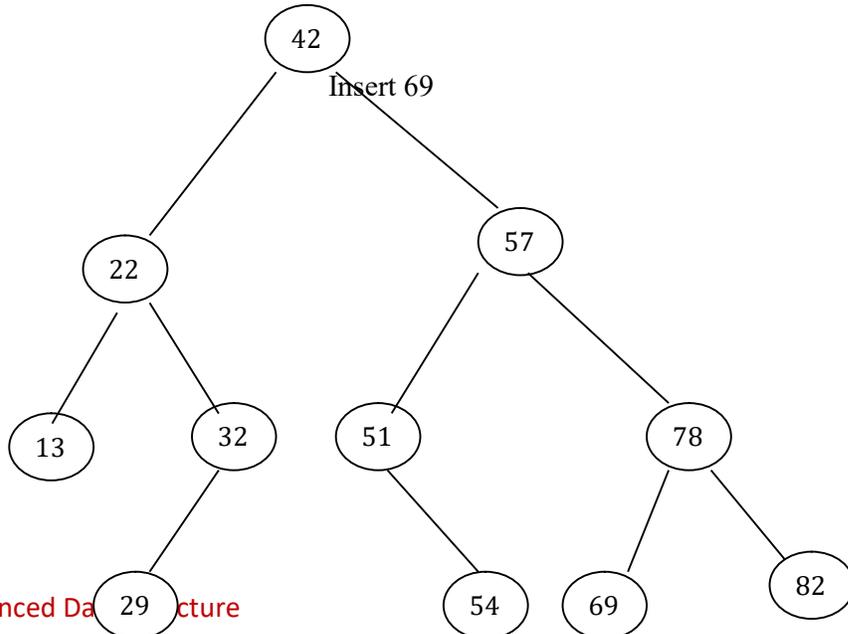
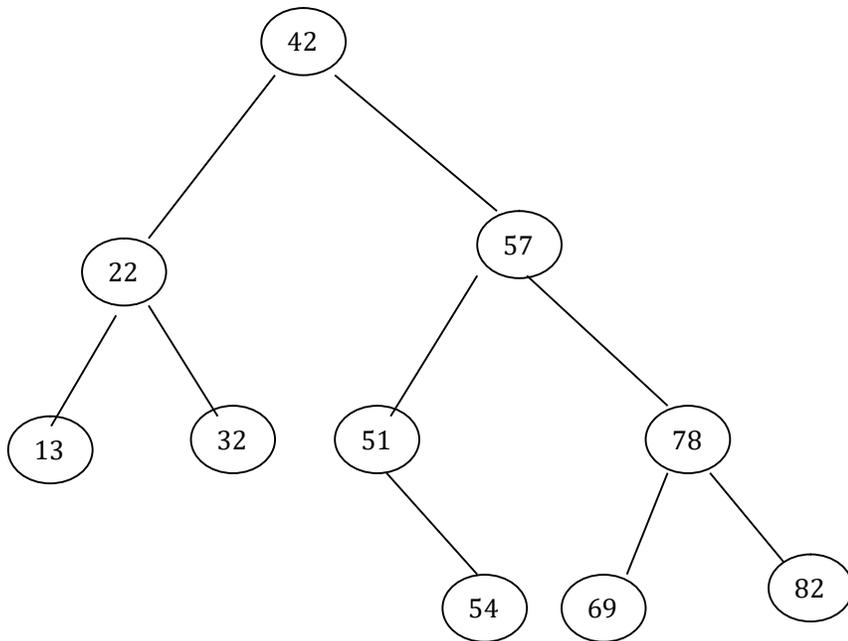
Insert 13



Insert 54



Insert 82



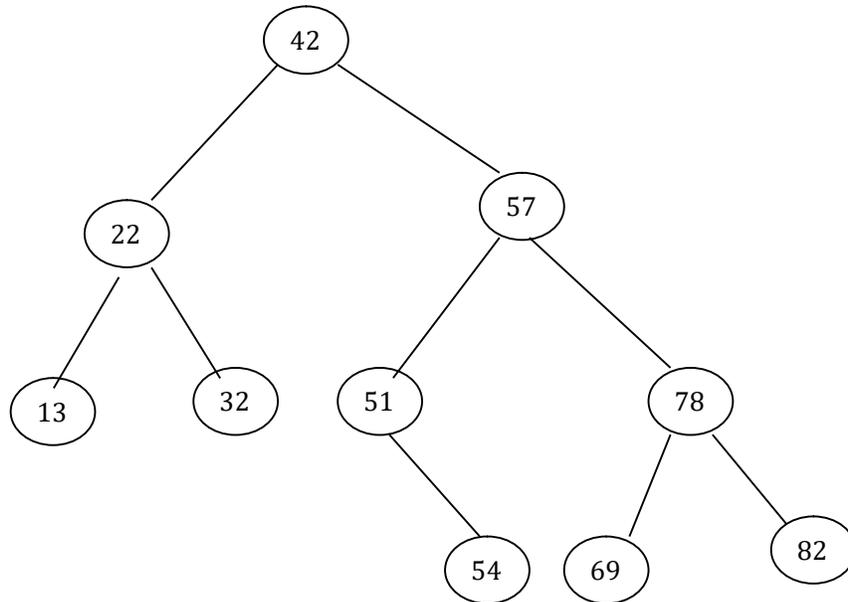
4.5.4 Insertion in Binary Search Tree:

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.

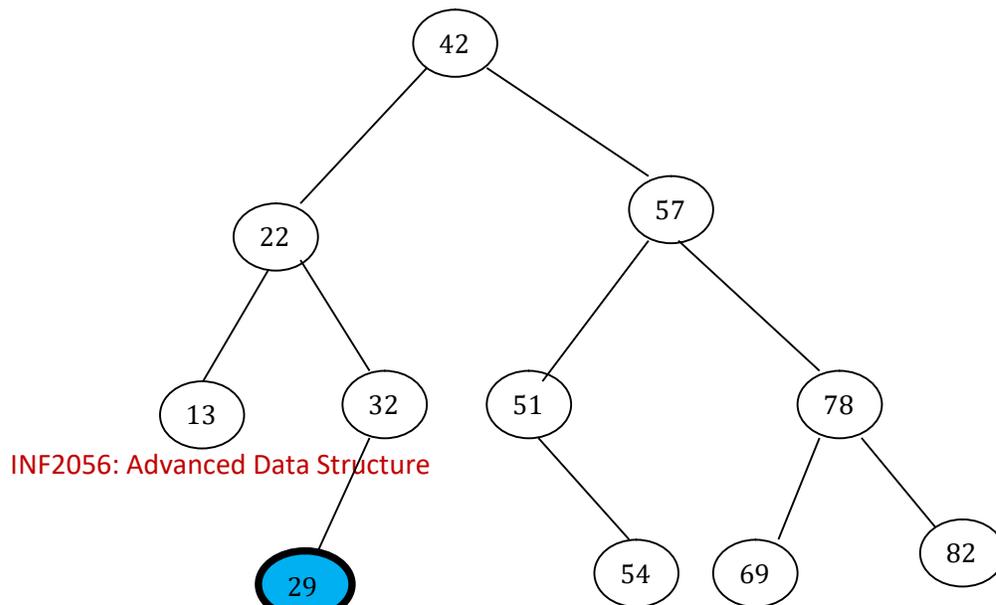
Insert (Tree, Insert_element)

- **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]
- **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 is exist. So, we will put this insert element in the left side. Now the resulting Binary search tree looks like-



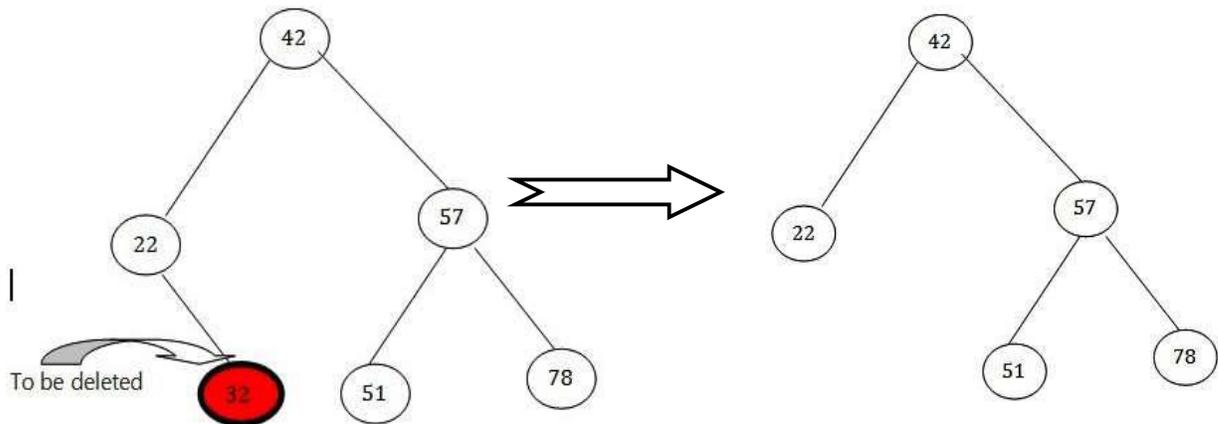
4.5.5 Delete an element from the Binary Search tree:

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.

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

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().



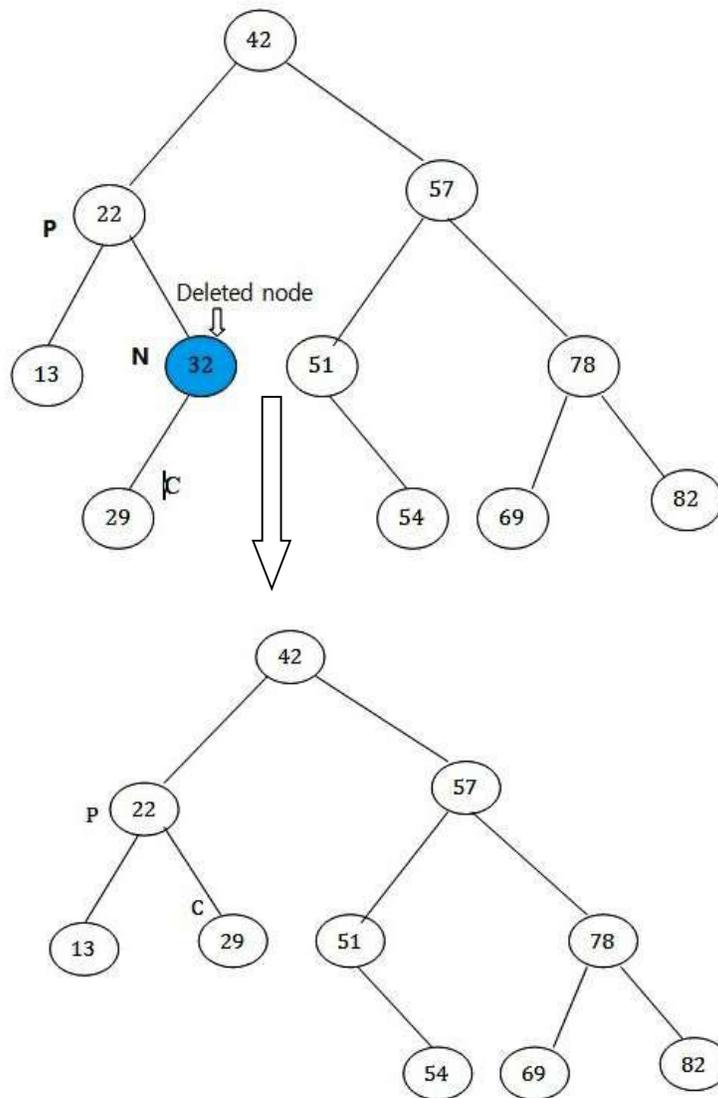
4.5.5.2 The node to be deleted has exactly one child

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.



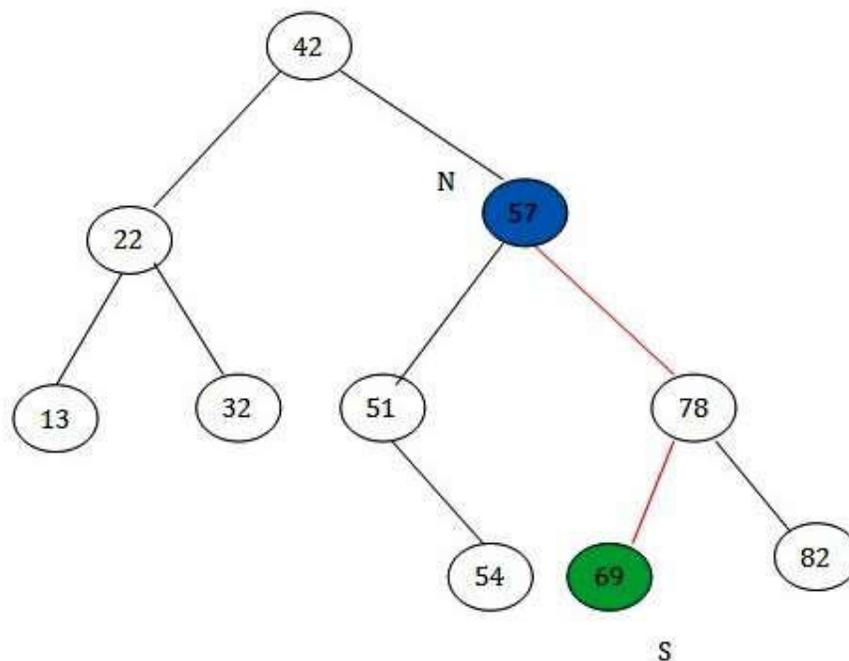
The Node 32 is to be deleted from the tree. Node 32 is right child of its parent 22. So the single child 29 will take the position of 32.

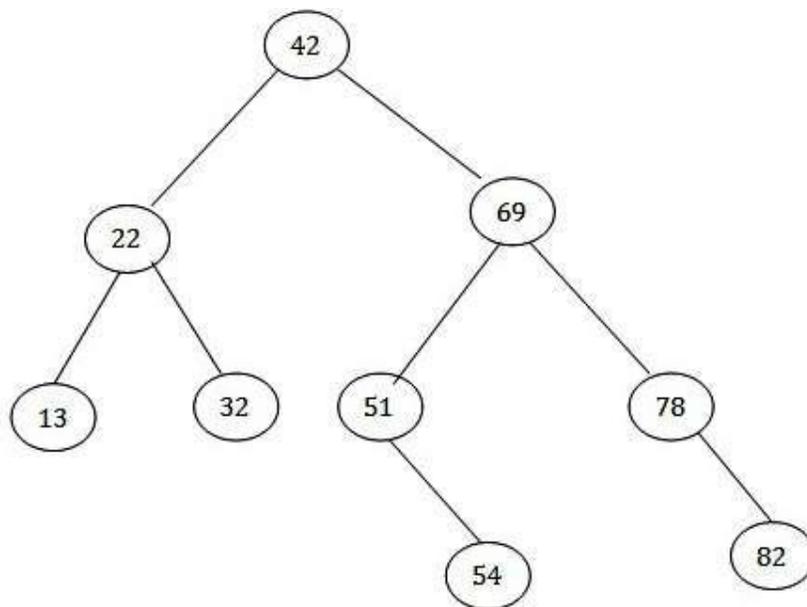
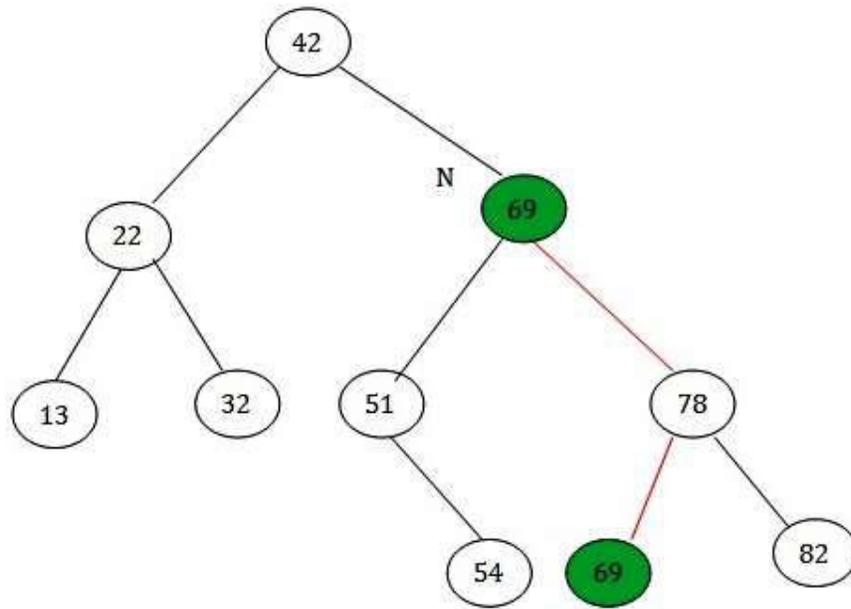
4.5.5.3 The node to be deleted has exactly two children.

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 inoder 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.





4.6 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 tasks in traversing- visiting the 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 Postorder 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 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() .

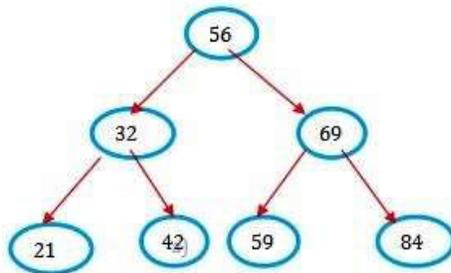
- When we delete a node from Binary search tree who have exactly two children, first we need to find out the In-order successor of the deleted node. The data of the In-order successor copied to the deleted node and then delete the In-order successor from the tree.

4.7 Questions and Answers:

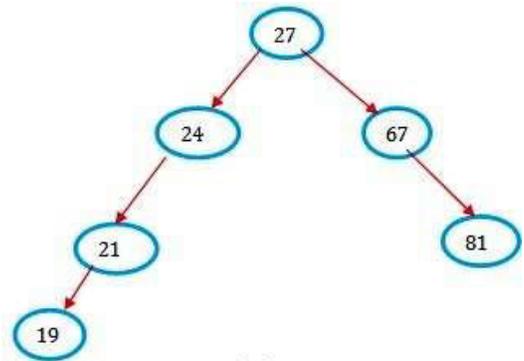
4.7.1 Multiple choice Question and Answer:

1. The number of edges from the root to the node is called _____ of the tree.
 - a) Height
 - b) Length
 - c) Depth
 - d) Width
2. The number of edges from the node to the deepest leaf is called _____ of the tree.
 - a) Height
 - b) Depth
 - c) Length
 - d) Width
3. What is a full binary tree?
 - a) Each node has exactly zero or two children

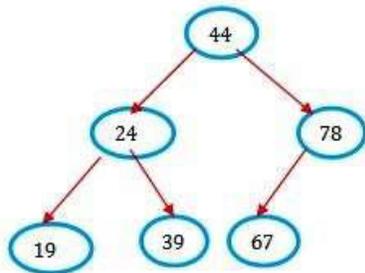
- b) Each node has exactly two children
 - c) All the leaves are at the same level
 - d) Each node has exactly one or two children
4. What is a complete binary tree?
- a) Each node has exactly zero or two children
 - b) A binary tree, which is completely filled, with the possible exception of the bottom 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
5. Which of the following is not a binary search tree:



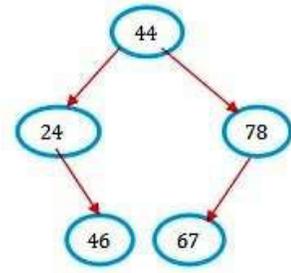
(I)



(II)

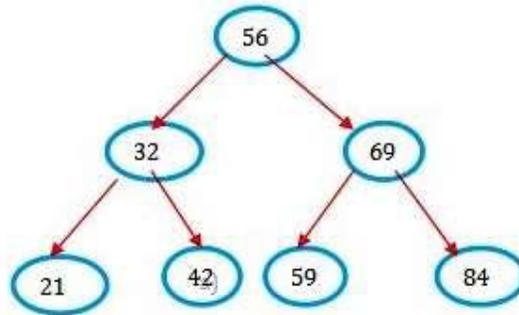


(III)



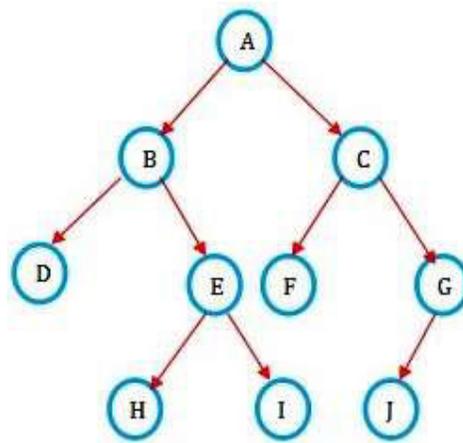
(IV)

- a) (I)
 - b) (II)
 - c) (III)
 - d) (IV)
6. If we delete 56 from the below Binary search tree after that what Parent --> Child pair does not occur in the tree?



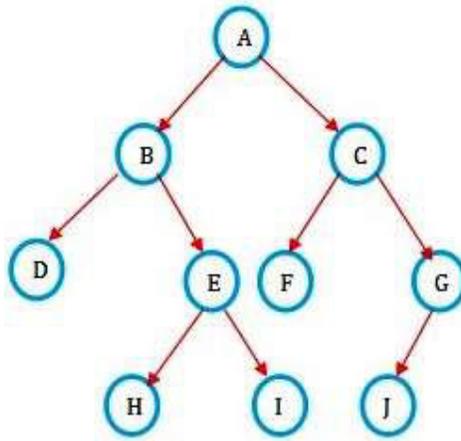
- a) 59-->69
- b) 32-->42
- c) 59-->84
- d) 32-->21

7. The In-order traversal of the below mention Binary tree:



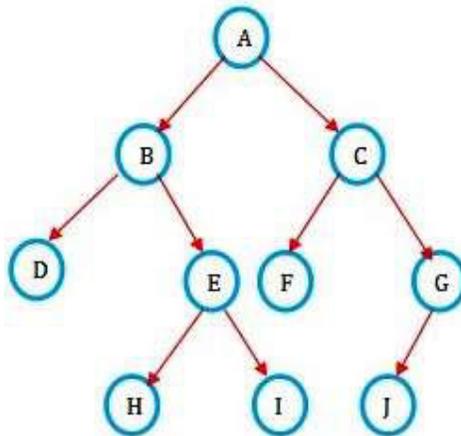
- a) A-B-D-E-H-I-C-F-G-J
- b) D-B-H-E-I-A-F-C-J-G
- c) D-H-I-E-B-F-J-G-C-A
- d) A-B-C-D-E-F-G-H-I-J

8. 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

9. The Postorder 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

10. 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

Solution:

1. The number of edges from the root to the node is called _____ of the tree.

Answer: c)

Explanation: The number of edges from the root to the node is called depth of the tree.

2. The number of edges from the node to the deepest leaf is called _____ of the tree.

Answer: a)

Explanation: The number of edges from the node to the deepest leaf is called height of the tree

3. What is a full binary tree?

Answer: a

Explanation: A full binary tree is a tree in which each node has exactly 0 or 2 children.

4. What is a complete binary tree?

Answer: c

Explanation: A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right is called complete binary tree. A Tree in which each node has exactly zero or two children is called full binary tree. A Tree in which the degree of each node is 2 except leaf nodes is called perfect binary tree.

5. Which of the following is not a binary search tree:

Answer: d). 46 is greater than root node value 44 which should be in the right side. We know that all the elements left side of root is lesser than it and the all the elements right side of root is greater than it.

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

Answer : c) 59 → 84

Explanation: In the above Binary search tree when we delete 56, the node value 59 will take the place of 56 because 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.

Left side of root will remain same.

So 59 is not parent of 84.

7. The In-order traversal of the below mention Binary tree:

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

Explanation:

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.

8. The Pre-order traversal of the below mention Binary tree:

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

Explanation:

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.

9. The Postorder traversal of the below mention Binary tree:

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

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

10. What is the specialty about the In-order traversal of a binary search tree?

Answer b) It traverses in an increasing order

4.8.1 Fill up the Blanks and Answer

1. For searching elements in linked list, the complexity is _____.
2. In a tree , the immediate predecessor of a node is called as _____.
3. Every tree must have a _____ node.
4. _____ is specially designated node that does not have any parent node.
5. The node of a tree which does not have a child is called as _____ Node.
6. 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.
7. In _____ traversal, the root node is visited first, then the left sub tree and finally the right subtree.
8. In _____ Traversal (Left-Right-Root), the left sub tree is visited first, then visit the right subtree , finally visit the root node. By Postorder traversal we can get the postfix expression of an expression.

Answer:

1. $O(n)$
2. Parent Node
3. Root
4. Root
5. Leaf
6. In-order
7. Pre-order
8. Post-order

4.8.2 True or False:

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

2. In a tree data structure, the total number of children of a node is called as Degree of that Node.
3. A Binary tree can have any number of children.
4. A Binary tree is strictly binary tree if each node in the tree is either a leaf node or should have exactly two children.
5. The full binary tree can be defined as a binary tree if all level has maximum number of nodes except the leaf nodes.

Answer:

1. False
2. True
3. False
4. True
5. True.

4.8.3 Short types question:

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?

4.8.4 Broad type questions:

1. Explain the two types of representation of Binary tree with diagram.

2. Explain various kinds of traversal techniques of binary tree with example.
3. Write the algorithm to search an element in a binary search tree?
4. Write down the insertion procedure of Binary search tree.
5. Write down the working procedure to delete an element from the binary search tree where-
 - The node to be deleted has no child i.e leaf node.
 - The node to be deleted has exactly one child.
 - The node to be deleted has exactly two children.
6. Write a function to check whether a binary tree is binary search tree or not.
7. Write a program to display all the leaf nodes of binary tree.

4.9 Suggested Readings:

1. **Data Structures Through C In Depth.** by S.K.Srivastava /Deepali Srivastava..
2. **Data Structures Using C** by Reema Thareja

UNIT 1: INTRODUCTION TO SEARCH TREES

CONTENTS

- 1.1 Introduction
- 1.2 Unit Objectives
- 1.3 Definition of Search Trees
- 1.4 Types of Search Trees
 - 1.4.1 Binary Search Tree (BST)
 - 1.4.2 AVL Tree
 - 1.4.3 B Tree
 - 1.4.4 (a,b)- Tree
 - 1.4.5 Red-Black Tree
 - 1.4.6 Splay Tree
 - 1.4.7 Ternary Search Tree
- 1.5 Balancing of Search Trees
- 1.6 Summing Up
- 1.7 Key Terms
- 1.8 Questions and Answers
- 1.9 Suggested Readings

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, DELETE. The time that an operation takes depends on the height h of the tree.

1.2 UNIT OBJECTIVES

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

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. 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.

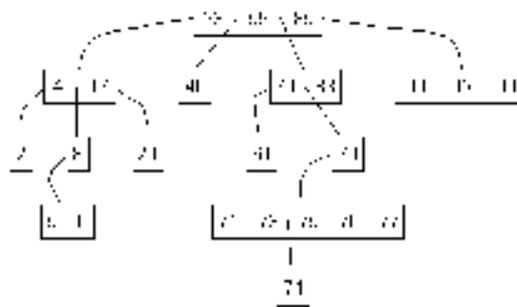


Fig. 1.1 An example of a search tree

1.4 TYPES OF SEARCH TREES

There are many different search trees 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

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 –

$$\text{Left subtree (keys)} < \text{node (key)} \leq \text{Right subtree (keys)}$$

A diagrammatic representation of a Binary Search Tree is given below-

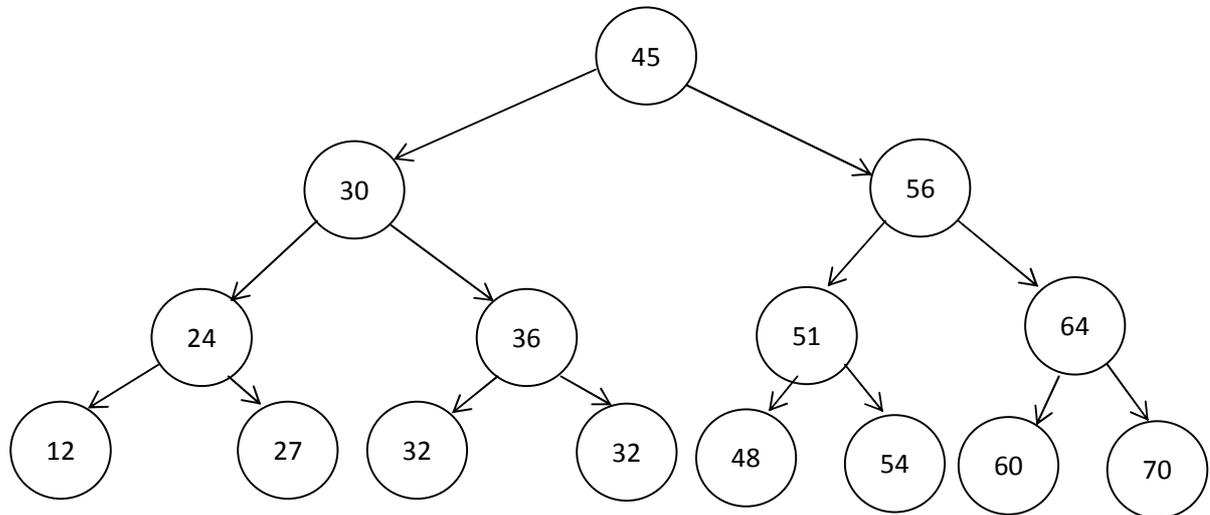


Fig. 1.3.1 example of a binary search tree

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
- $h_L - h_R \leq 1$, 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.

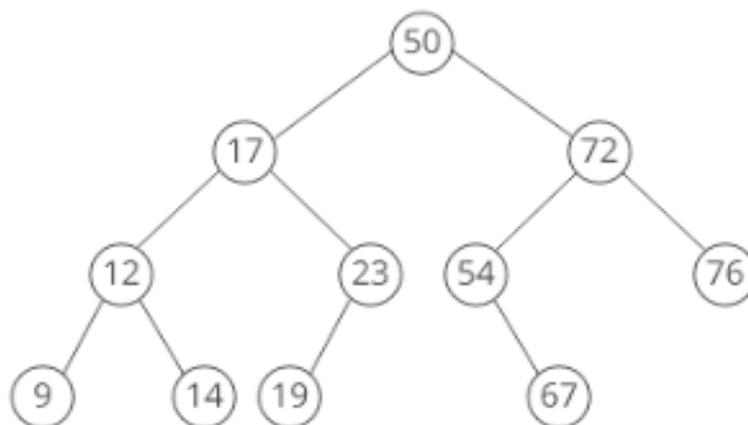


Fig. 1.3.2.1 balanced binary search tree

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

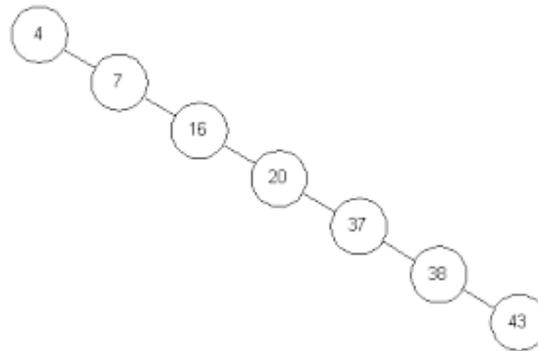


Fig. 1.3.2.2 unbalanced binary search tree

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

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 preserve 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.

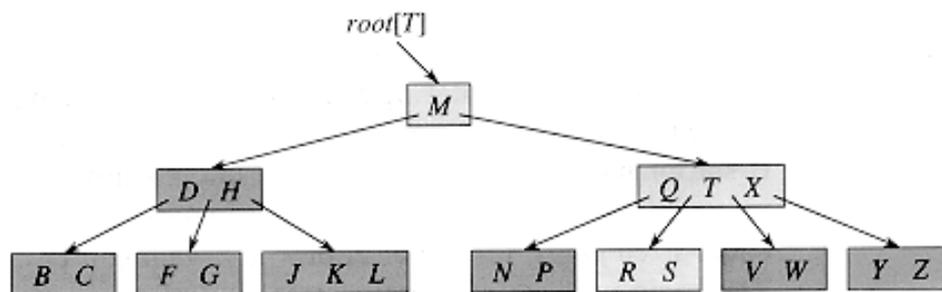


Fig.1.2.3.1 A B-tree whose keys are the consonants of English. An internal node x containing $n[x]$ keys has $n[x] + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

B-trees generalize binary search trees in a natural manner. Figure 1.3.3.1 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.

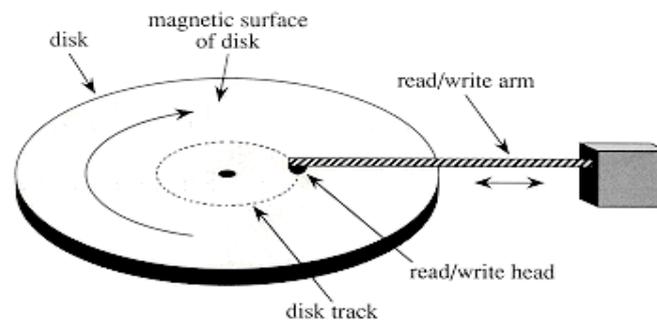


Fig.1.3.3.2 A typical disk drive.

Figure 1.3.3.2 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:

- the number of disk accesses, and
- 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 $\text{DISK-READ}(x)$ 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 $\text{DISK-READ}(x)$ requires no disk accesses; it is a "noop.") Similarly, the operation $\text{DISK-WRITE}(x)$ 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 $\text{DISK-READ}(x)$
- 4 operations that access and/or modify the fields of x
- 5 $\text{DISK-WRITE}(x)$ ▶ Omitted if no fields of x were changed.
- 6 the operations that access but do not modify fields of x
- 7 ...

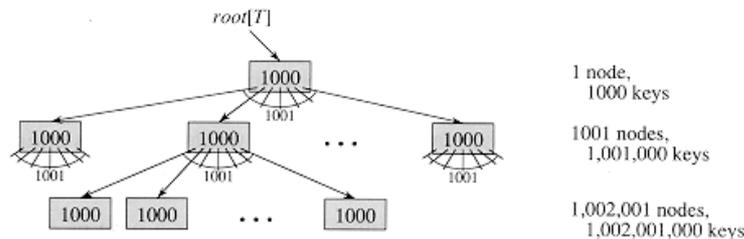


Fig.1.3.3.3 A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x .

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. Fig. 1.2.3.3 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 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: $key_1[x] \leq key_2[x] \leq \dots \leq key_n[x]$, 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 $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$ 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

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \dots \leq key_n[x] \leq k_{n[x]+1}$$
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 $t \geq 2$ 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.
 - 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 $2t - 1$ keys.

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a **2-3-4 tree**. 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.

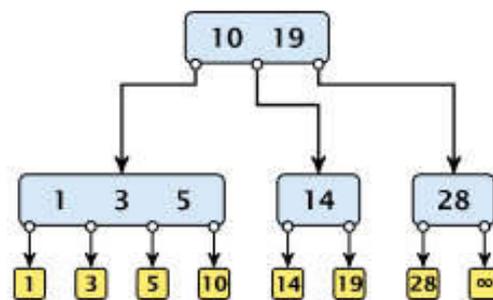


Fig. 1.3.4.1 an example of a (a,b) Tree

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.

Example: to insert 10 into the tree shown below-

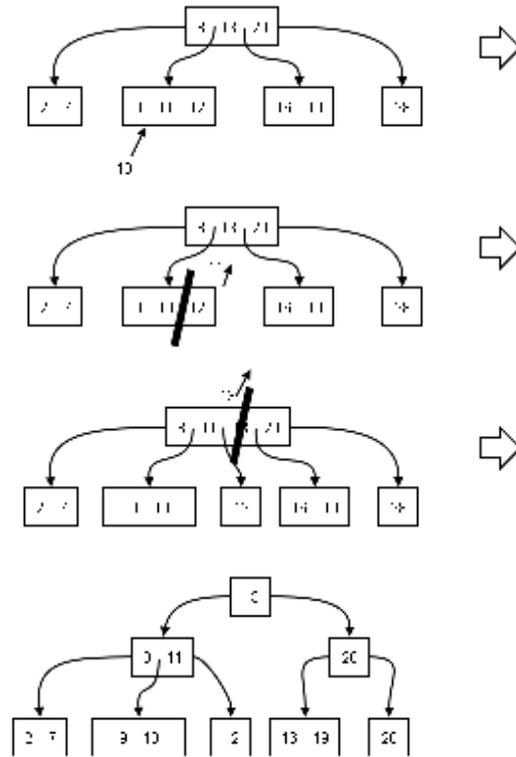


Fig. 1.3.4.2 an example of insertion into a (a,b) Tree

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:

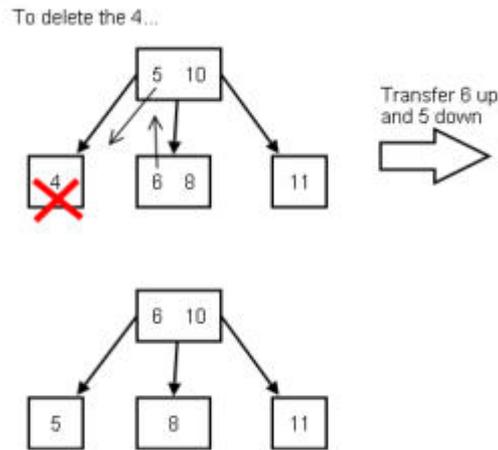


Fig. 1.4.2.3 an example of deletion from a (a,b) Tree

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.

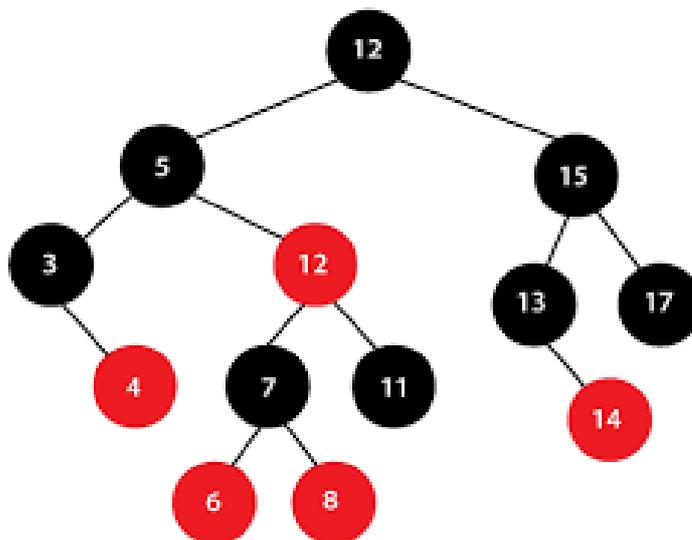


Fig.1.3.5.1 Red Black Tree

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.

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.

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

n is the total number of nodes in red black 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 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.

1.4.7 TERNARY SEARCH TREES

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.

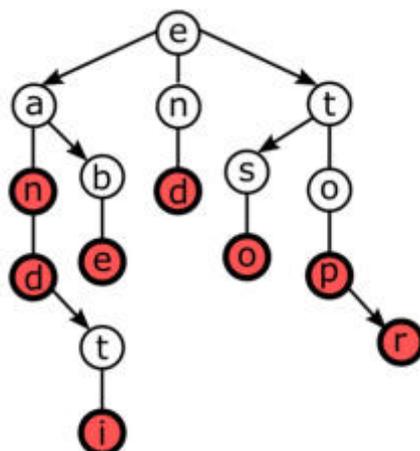


Fig. 1.3.7.1 A Ternary Search Tree

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 s_L 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 $s+c$: $s_L < s+c$ (or, to put it in another way, the next character in s_L is lexicographically less than c : if $|s|=m$, $s_L[m] < c$).
2. All keys s_R stored in the right sub-tree of N starts with s , are longer than s , and lexicographically greater than $s+c$: $s_R > s+c$.
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.

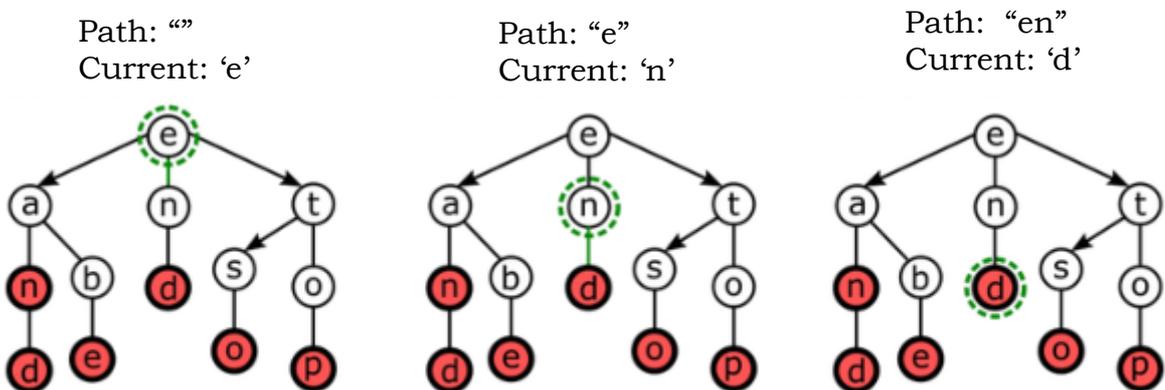
For instance, let's take the root of the tree:

- 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'.

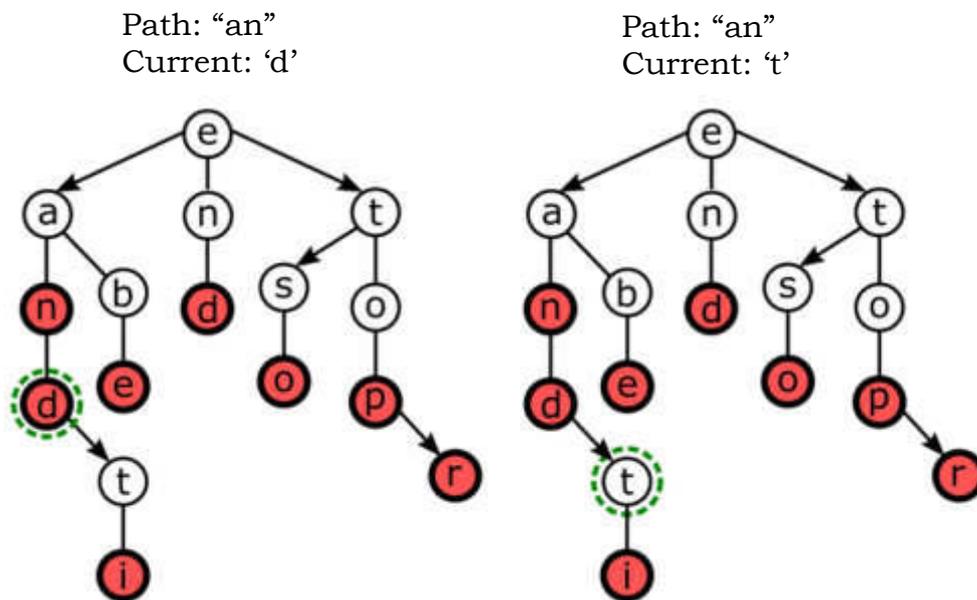
When we traverse a *Ternary Search Tree*, we keep track of a "search string", as we do with tries: for each node N, it's the string that could be stored in N, and it's determined by the path from the root to N. The way we build this search string is, however, very different with respect to tries.

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 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".

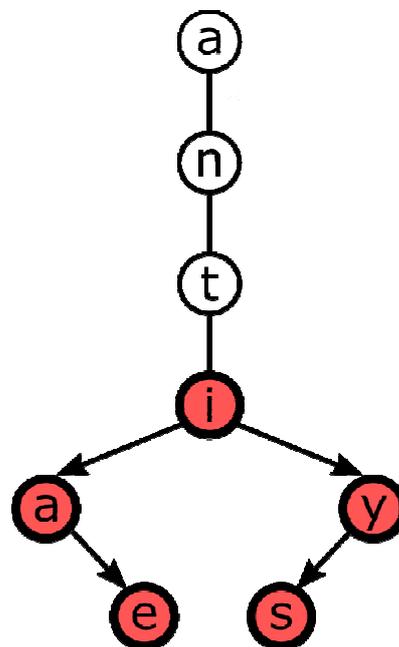
Although Ternary Search Tree is an alternative to tries, but the question arises that how efficient this data structure than tries? So we

will discuss and space and time complexity of Ternary Search Tree to get a better understanding of the concept.

SPACE COMPLEXITY

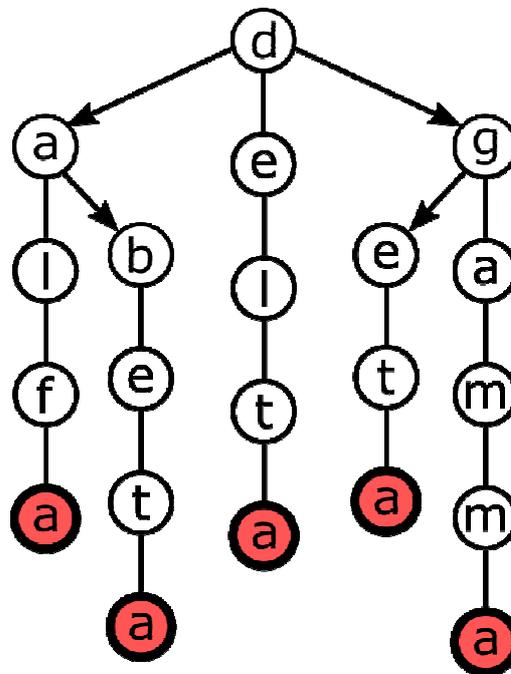
So, the question now arises: how many links (and nodes) are created for such *Ternary Search Tree*? To answer that, suppose we want to store n keys whose average length is w . Then we can say that:

1. The minimum number of links needed will 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 $w=4$.



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.

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.

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

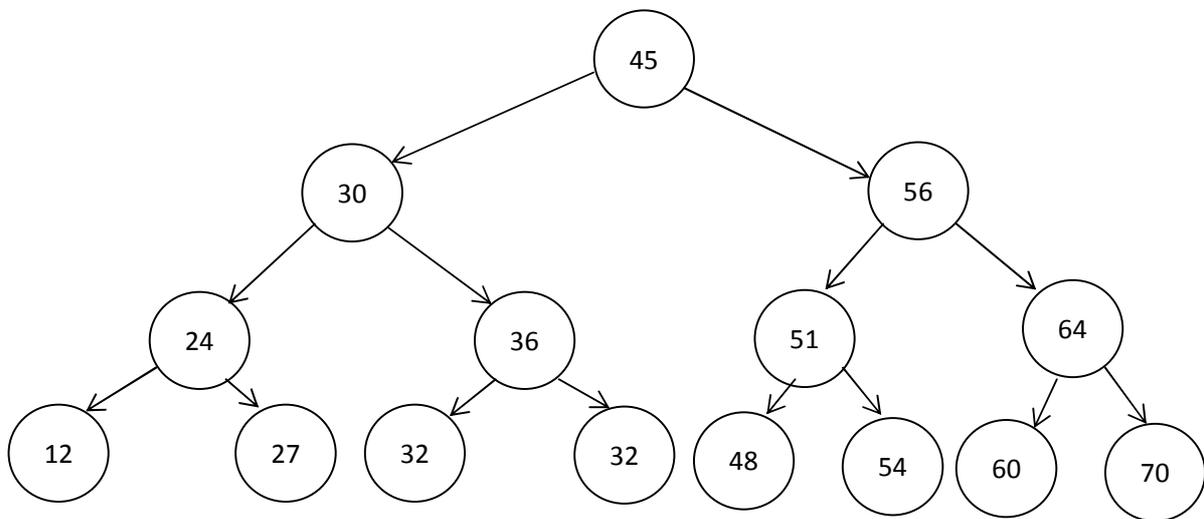


Fig. 1.4.1 Binary Search Tree

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 having 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.

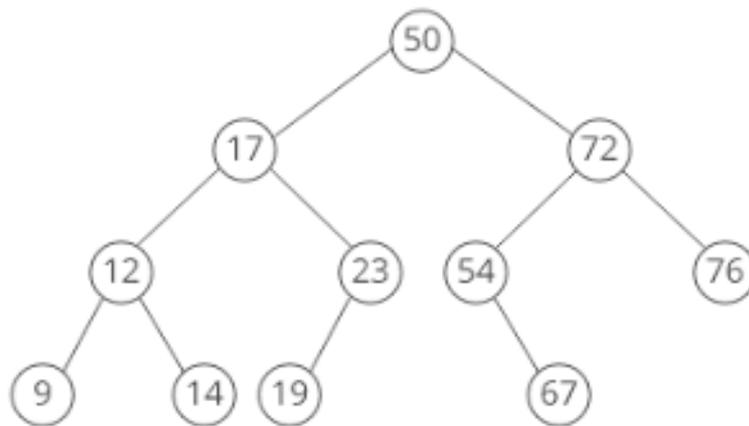


Fig. 1.4.2 Balanced Binary Search Tree

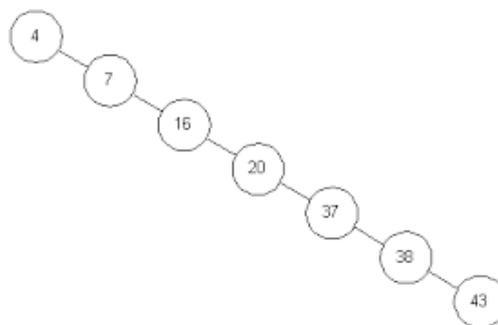


Fig. 1.4.3 Unbalanced Binary Search Tree

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 ensure that search time strict to $O(\log(n))$ of complexity.

1.7 KEY TERMS

- **Root:**In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.
- **Edge:** In a tree data structure, 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:** In a tree data structure, the node which is a predecessor of any 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. Parent node can also be defined as "**The node which has child / children**".
- **Child:** In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple 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. In a tree, all the nodes except root are child nodes.
- **Siblings:** In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.

- **Leaf:** 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.
- **Internal Nodes:** In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.
- **Degree:** In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'.
- **Level:** In a tree data structure, 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... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).
- **Height:** In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. 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'**.
- **Depth:** In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'**.

- **Path:** In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path.
- **Sub Tree:** In a tree data structure, each child from a node forms a sub-tree recursively. Every child node will form a sub-tree on its parent node.

1.8 QUESTIONS AND ANSWERS

Multiple Choice Questions:

1. 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
2. What is the speciality about the inorder 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
3. What are the worst case and average case complexities of a binary search tree?
 - a) $O(n)$, $O(n)$
 - b) $O(\log n)$, $O(\log n)$
 - c) $O(\log n)$, $O(n)$
 - d) $O(n)$, $O(\log n)$
4. The minimum height of an AVL Tree with n node is

- a) $\text{Ceil}(\log_2(n+1))$
 - b) $1.44 \log_2 n$
 - c) $\text{Floor}(\log_2(n+1))$
 - d) $1.64 \log_2 n$
5. 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.
6. 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
7. 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
8. 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 $\log(\text{nodes})$ time complexity is needed
9. 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 at most, takes $O(j \log h)$ time complexity
 - c) sequence of operations with h nodes can take $O(\log h)$ time complexity
 - d) splay trees are unstable trees
10. How many child nodes does each node of Ternary Tree contain?
- a) 4
 - b) 6
 - c) 5
 - d) 3

Answers: 1. d., 2.b., 3. d.,4. c.,5. b.,6. a.,7. c., 8. a., 9. b.,10. d.

Fill in the following blanks:

1. To arrange a binary search tree in ascending order, we need _____ order traversal only.
2. Consider the binary search tree with n elements. The time required to search given element is _____.
3. In _____ balance factor of a node is the difference between left sub-tree and right sub-tree.
4. The maximum height of an AVL tree with p nodes is _____.
5. B-tree of order n is a order- n multiway tree in which each non-root node contains _____ keys.
6. A B-tree of order 4 and of height 3 will have a maximum of _____ keys.
7. 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.
8. In a Red-Black Tree, if a node is red, its child must be _____.
9. Self-adjusting binary search tree is called _____.

10. Each node of ternary tree contains _____ number of child nodes.....

Answers: 1.In, 2. $\theta(\log n)$, 3. AVL Tree, 4. $\log(p)$, 5. at least $(n - 1)/2$, 6. 265, 7. black depth, black height.,8. Black, 9.Splay Tree, 10. 3.

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?
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#.

1.9 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 Algorithmic Puzzles* 5th ed. Edition, The Career Monk.

UNIT 2: AVL TREES AND RED-BLACK TREES

CONTENTS

- 2.1 Introduction
- 2.2 Unit Objectives
- 2.3 AVL Tree
 - 2.3.1 Introduction
 - 2.3.2 AVL Tree Rotation
 - 2.3.3 Search Operation in AVL Tree
 - 2.3.4 Insertion Operation in AVL Tree
 - 2.3.5 Deletion Operation in AVL Tree
 - 2.3.6 Complete Program of AVL Tree
- 2.4 Red-Black Tree
 - 2.4.1 Introduction
 - 2.4.2 Insertion Operation in Red-Black Tree
 - 2.4.3 Deletion Operation in Red-Black Tree
 - 2.4.4 Complete Program of Red Black Tree
- 2.5 Summing Up
- 2.6 Key Terms
- 2.7 Questions and Answers
- 2.8 Suggested Readings

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.

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
- $h_L - h_R \leq 1$, 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.

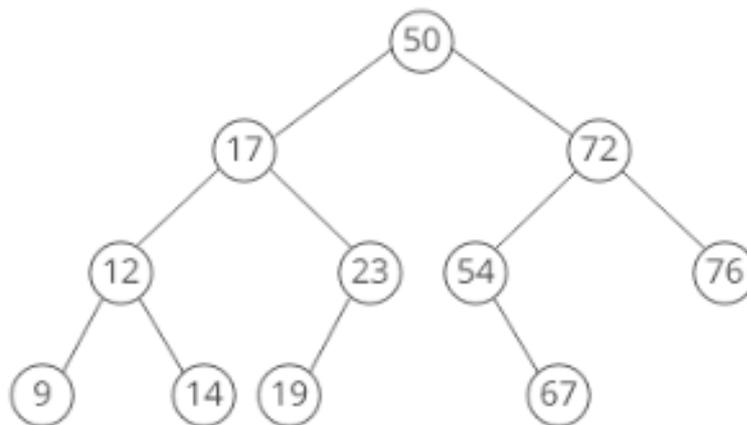


Fig. 2.3.1 balanced binary search tree

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

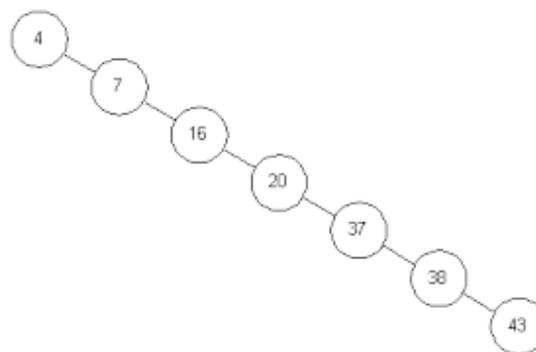


Fig. 2.3.2 unbalanced binary search tree

In the above figure 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 fig. 2.3.2, performing these operation takes $O(n)$ time, where 'n' denotes the number of nodes in the tree.

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 to $O(\log n)$. 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-

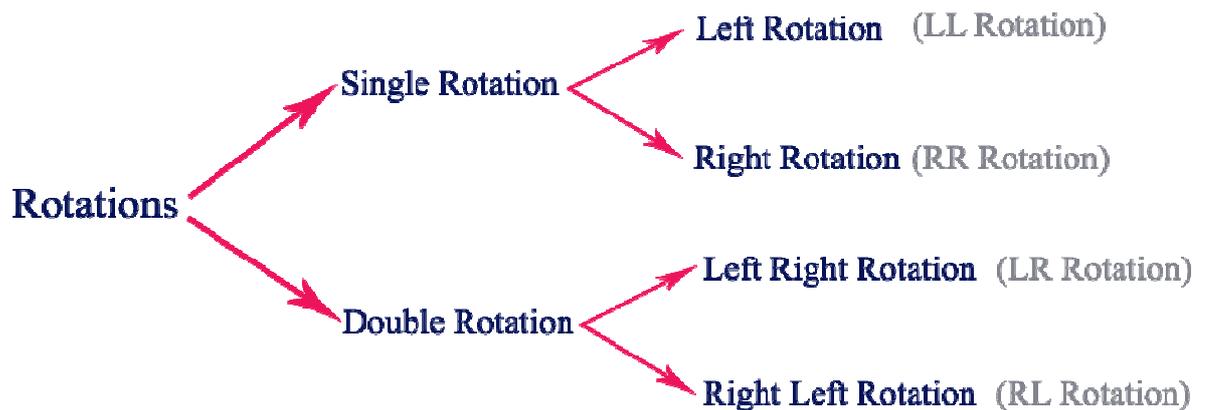
```
typedef struct AVL
{
    int data;
    struct AVL *lnode;
    struct AVL *rnode;
    int height;
}AVL;
```

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 rotationoperations 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.



There are four rotations and they are classified into two types.

Fig. 2.3.2.1 types of rotation of AVL Tree

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-

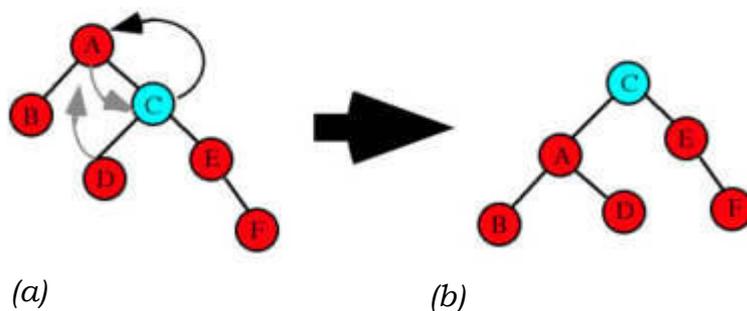


Fig. 2.3.2.2 Left rotation of AVL Tree

From the diagram (a) it is clear that the tree is unbalanced 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-

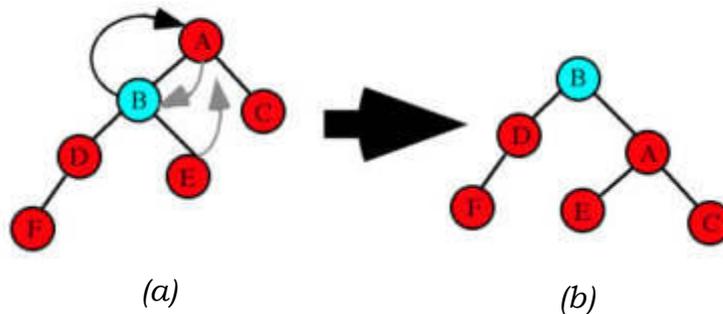


Fig. 12.3.2.3 Right rotation of AVL Tree

From the above diagram (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 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.

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*.

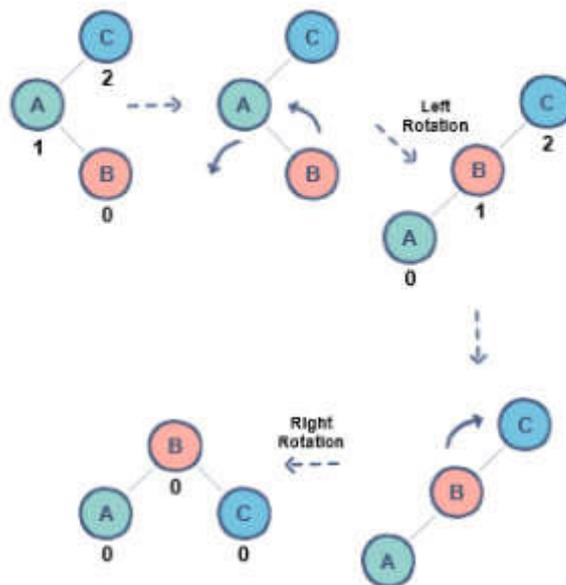


Fig. 2.3.2.4 Left Right rotation of AVL Tree

In the given example, **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.

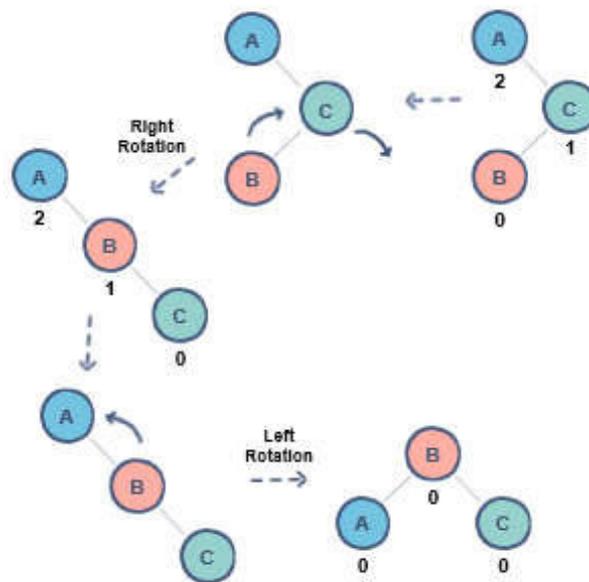


Fig.2.3.2.5 Right Left rotation of AVL Tree

In the given example, 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 Order Traversal

```
void inorder(AVL *root)
{
    if(root!=NULL)
    {
        inorder(root->lnode);
        printf("%d",root->data);
        inorder(root->rnode);
    }
}
```

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 $BF(Z) \neq 0$ (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.

2.3.6 COMPLETE PROGRAM OF AVL TREE

The complete C program for AVL tree is demonstrated below (the program is tested in Linux Environment)-

```
#include<stdio.h>
#include<stdlib.h>

typedef struct AVL
{
    int info;
    struct AVL *lnode,*rnode;
    int ht;
}AVL;

AVL *insert(AVL *,int);
AVL *Delete( AVL *,int);
void preorder(AVL *);
void inorder(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;
    int x,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);
        }
        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)
        { //delete its inorder successor
            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)
{
    int lh,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);
    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)
{
    int lh,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;
    return(lh-rh);
}

void preorder(AVL *root)
{
    if (root!=NULL)
    {
        printf("%d(Bf=%d) ", root->info, BF(root));
        preorder(root->lnode);
        preorder(root->rnode);
    }
}

void inorder(AVL *root)
{
    if (root!=NULL)
    {
        inorder(root->lnode);
        printf("%d(Bf=%d) ", root->info, BF(root));
        inorder(root->rnode);
    }
}
```

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.

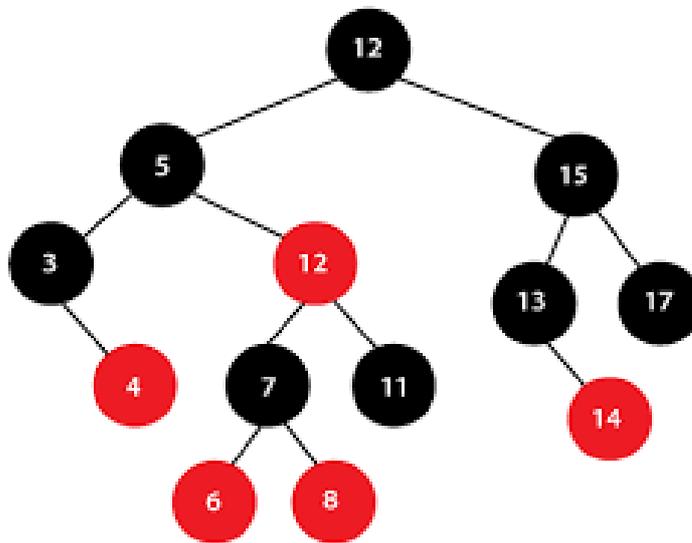


Fig.2.4.1.1 Red Black Tree

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.

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.

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

n is the total number of nodes in red black tree

2.4.2 INSERTION OPERATION IN RED-BLACK TREE

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 newNode as 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.

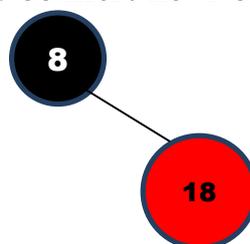
Insert (8)

The tree is Empty. So insert newNode as Root node with black colour.



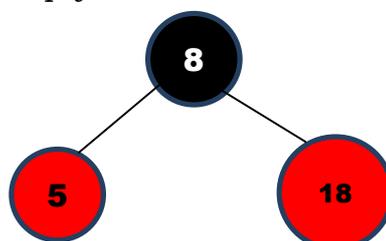
Insert (18)

The tree is not Empty. So insert newNode with red colour.



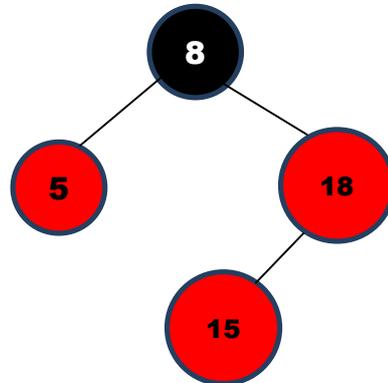
Insert (5)

The tree is not Empty. So insert newNode with red colour

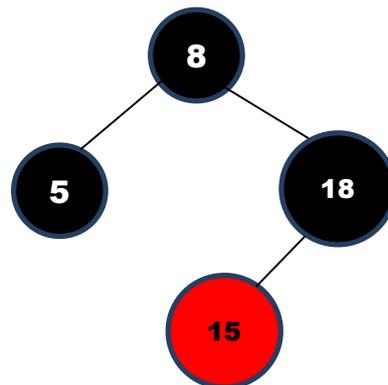


Insert (15)

The tree is not Empty. So insert newNode with red colour



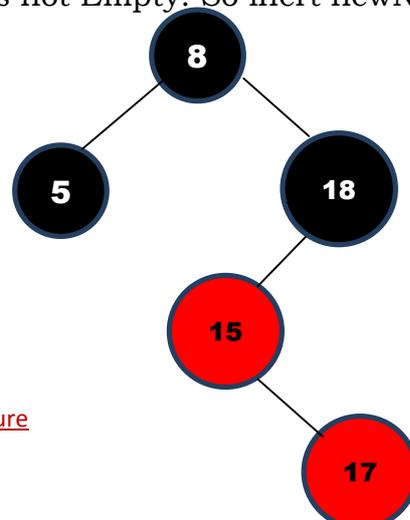
In the above diagram two consecutive red nodes (18 and 15). The newNode's parent and sibling colour is red and parent's parent is root node. So we have to use RECOLOR to make red black tree.



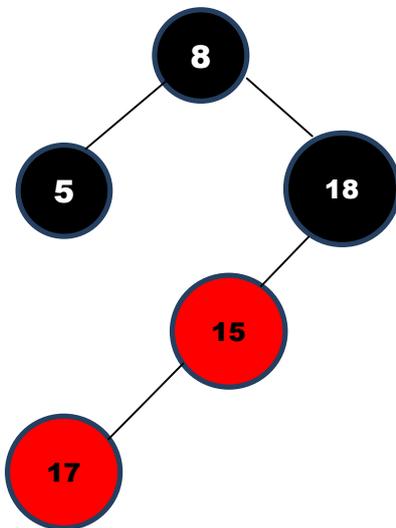
After recolor operation, the tree is satisfying all Red Black Tree properties.

Insert (17)

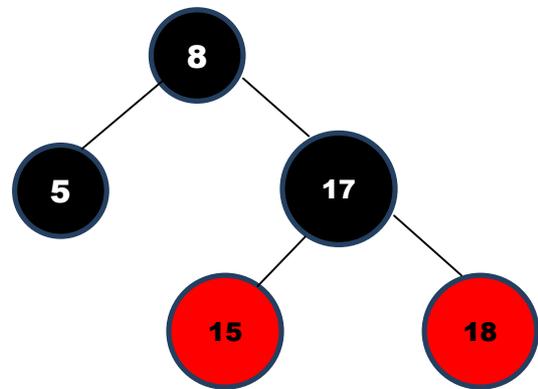
The tree is not Empty. So insert newNode with red colour



In the above diagram two consecutive red nodes (15 and 17). The newNode's parent sibling is NULL. So we need rotation. Here we need LR rotation and RECOLOR.



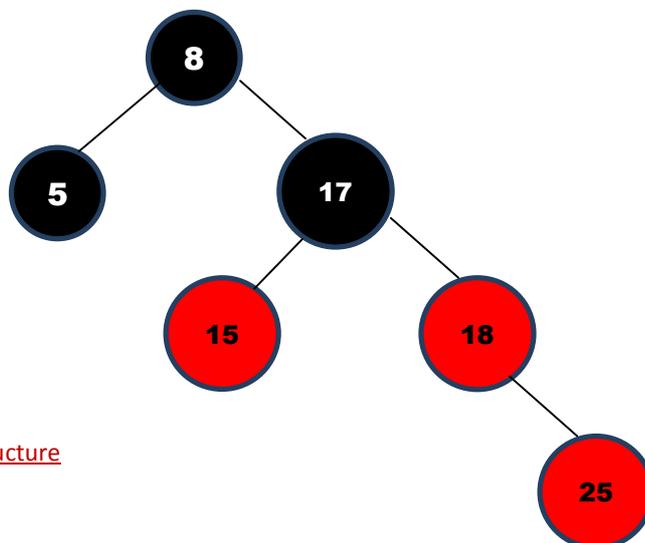
After Left Rotation



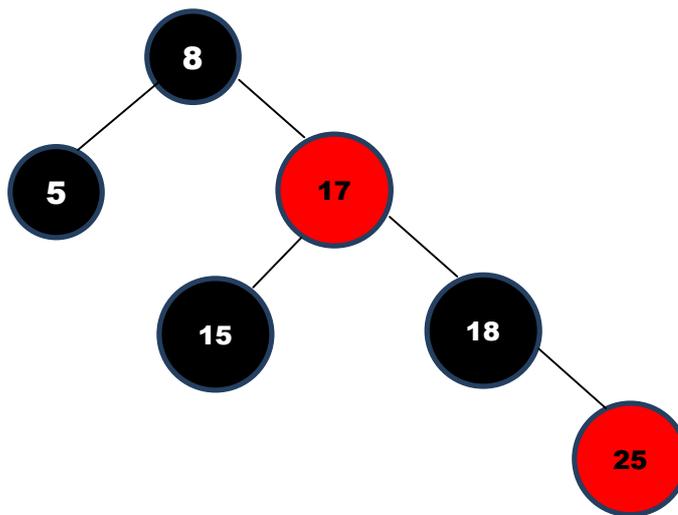
After Right Rotation and Recolour

Insert (25)

The tree is not Empty. So insert newNode with red colour



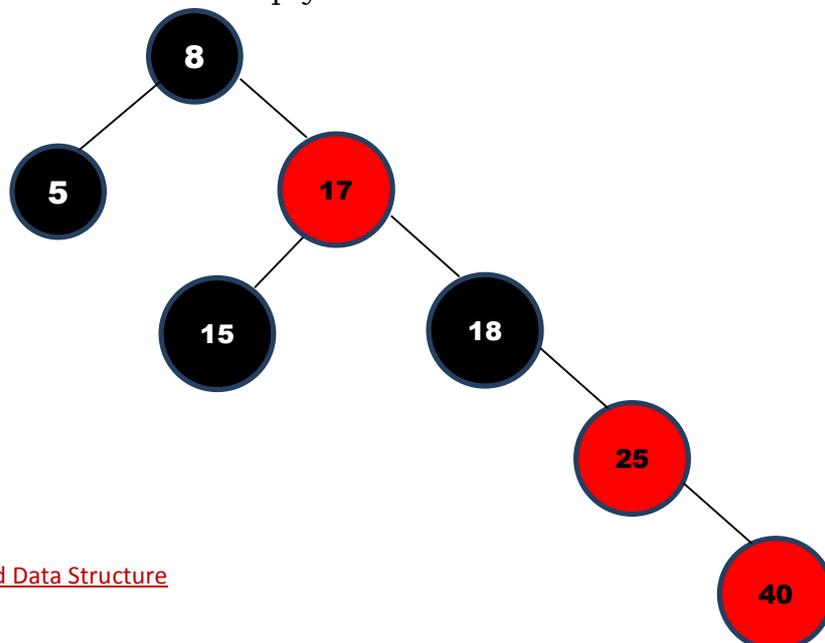
In the above diagram two consecutive red nodes (18 and 25). The newNode's parent sibling colour is Red and parent's parent is not root node. So we use RECOLOR and Recheck.



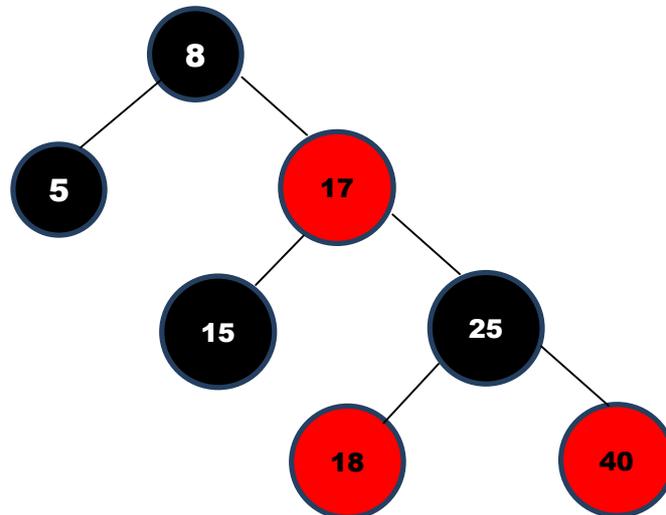
After Recolour operation the tree is satisfying all Red Black Tree properties.

Insert (40)

The tree is not Empty. So insert newNode with red colour



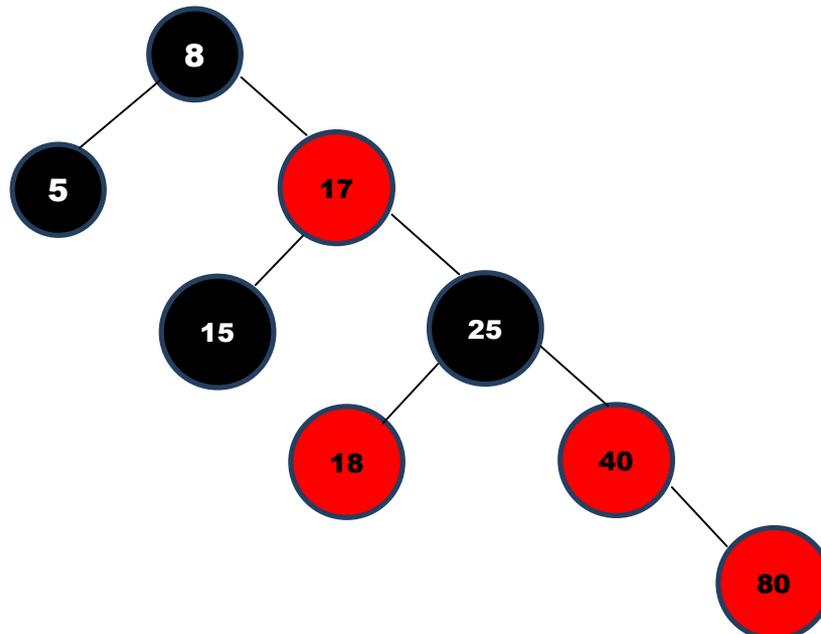
In the above diagram two consecutive red nodes (25 and 40). The newNode's parent sibling is NULL. So we need RECOLOR and Recheck. Here we use LL Rotation and Recheck.



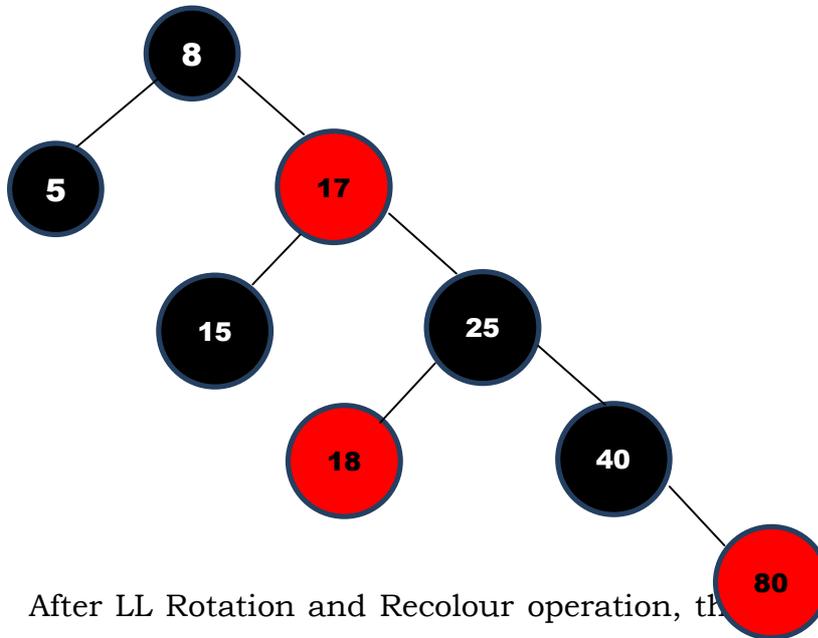
After LL Rotation and Recolour operation, the tree is satisfying all Red Black Tree properties.

Insert (80)

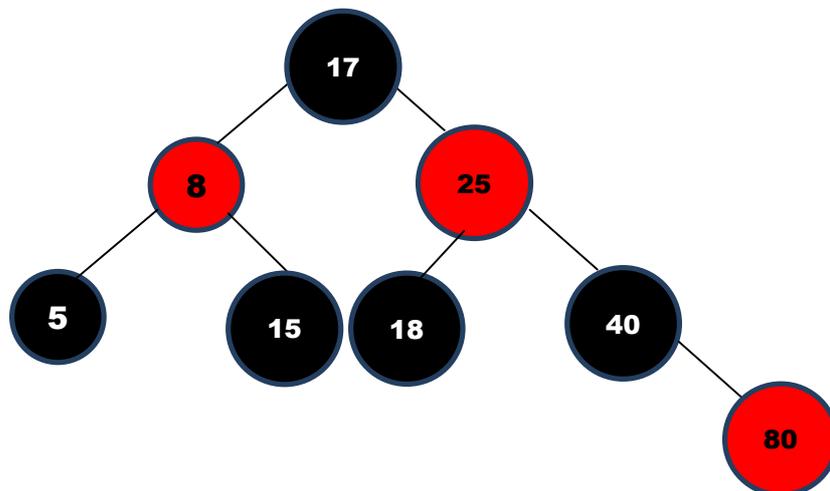
The tree is not Empty. So insert newNode with red colour



In the above diagram two consecutive red nodes (40 and 80). The newNode'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 FROM 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 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-

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**, perform recoloring, and recur 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 subcases.
 - (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).

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(int val) {
    RBtree *newnode;
    newnode = (RBtree *)malloc(sizeof(RBtree));
    newnode->info = val;
    newnode->color = RED;
    newnode->link[0] = newnode->link[1] = NULL;
    return newnode;
}

/* Insert an node */
void insert(int val) {
    RBtree *stack[98], *ptr, *newnode, *xPtr, *yPtr;
    int dir[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];
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(int val) {
    RBtree *stack[98], *ptr, *xPtr, *yPtr;
    RBtree *pPtr, *qPtr, *rPtr;
    int dir[98], ht = 0, diff, i;
    enum nodeColor 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;
        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;

    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;

    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--;
}
}
```

```
/* Inorder traversal of the tree */
```

```
void inorderTraversal(RBtree *node) {
    if (node) {
        inorderTraversal(node->link[0]);
        printf("%d ", node->info);
        inorderTraversal(node->link[1]);
    }
    return;
}
```

```
/* Main Block */
```

```
int main() {
    int ch, 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");
}
return 0;
}
```

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.

2.6 KEY TERMS

Please refer to point no 1.7 of unit 1.

2.7 QUESTIONS AND ANSWERS

Multiple Choice Questions

1. 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

2. 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

3. 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.

4. 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
5. 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.
6. 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
7. 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.
8. 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.
9. 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
10. 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.

Answers: 1(a), 2 (b), 3 (b), 4 (a), 5 (b), 6 (a), 7(a), 8 (c), 9 (d), 10 (c)

Fill in the following blanks

1. In _____, balance factor of a node is the difference between height of left and right sub-tree.
2. There are _____ numbers of rotations are available at AVL tree.
3. The worst case running time to search for an element in an AVL tree with $n \times 2^n$ element is _____.
4. 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.

5. The worst case time complexity of AVL tree is better in comparison to binary search tree for _____, _____, _____ Operations.
6. In a Red Black Tree the restrictions are imposed to get _____ time complexity.
7. While inserting into _____ tree, insertions are done at a leaf and will replace an external node with an internal node with two external children.
8. The Red Black Tree is a binary search tree whose leaves are _____ nodes.
9. 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.
10. A Red Black tree guarantees _____ time for searching.

Answers: .1. AVL tree, 2. 4, 3. $O(\log n)$, 4. Average Path Length, 5. Search, Insert, Delete, 6. logarithmic, 7. Red Black, 8. external, 9. black depth black height, 10. $O(\log n)$.

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?

2.8 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

CONTENTS

- 1.1 Introduction
- 1.2 Unit Objectives
- 1.3 Multi way Search Tree
 - 1.3.1 Introduction
 - 1.3.2 Searching in a Multi way search tree
 - 1.3.3 Insertion into a Multi way search tree
 - 1.3.4 Deletion in Multi way search tree
- 1.4 2-3 Trees
 - 1.4.1 Introduction
 - 1.4.2 Searching Operation in an 2-3 Tree
 - 1.4.3 Insertion Operation in an 2-3 Tree
 - 1.4.4 Deletion operation in an 2-3 Tree.
- 1.5 Splay Tree
 - 1.5.1 Introduction
 - 1.5.2 Rotations in Splay Tree
 - 1.5.3 Insertion Operation in Splay Tree
 - 1.5.4 Deletion Operation in Splay Tree
- 1.6 Summing Up
- 1.7 Key Terms
- 1.8 Questions and Answers
- 2.1 Suggested Readings

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.

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 jth key.
- The keys in the last m-j children are higher than the jth key.

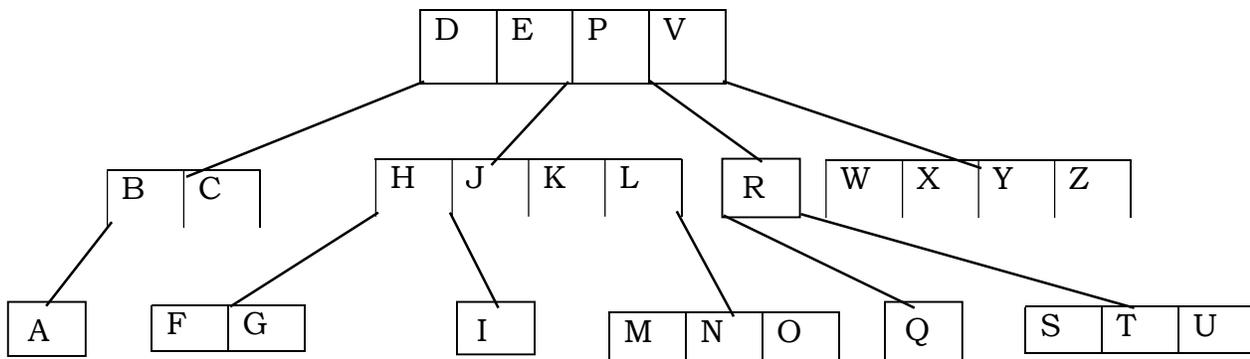


Fig. 3.3.1.1 A multi way tree of order 5

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 in sequential order. Multi way search tree provides fast information retrieval and update.

However, they also have some 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.
- All node except the root node, must have at least $\lceil M/2 \rceil - 1$ keys and at most $M-1$ keys.

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 $M_1, M_2, M_3, \dots, M_k$. Then in total 4 cases are possible to deal with this scenario, these are:

- If $N < M_1$, then we need to recursively traverse the left sub-tree of M_1 .
- If $N > M_k$, then we need to recursively traverse the right sub-tree of M_k .
- If $N = M_i$, for some i , then we are done, and can return.
- Last and only remaining case is that when for some i we have $M_i < N < M_{i+1}$ then in this case we need to recursively traverse the sub-tree that is present in between M_i and M_{i+1} .

A pictorial representation of an M-way search tree is shown below-

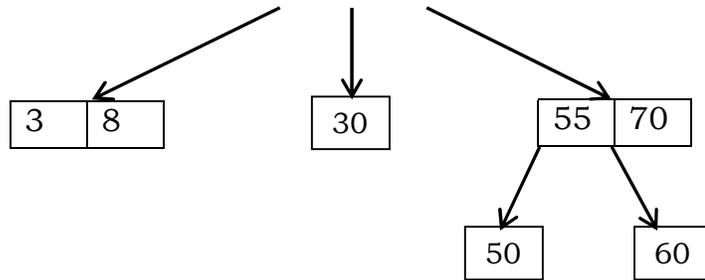


Fig. 3.3.2.1 an M-way search tree (it is a 3-way search tree)

From the fig. 3.3.2.1, 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

```

typedef struct node
{
    int num;
    int val[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.

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

The coding for searching in M-way search tree is given below

```
mTree *search(int val, 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);
    }
}

int searchnode(int val, 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])
            return 1;
        else
            return 0;
    }
}
```

3.3.3 INSERTION INTO A MULTIWAY 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.

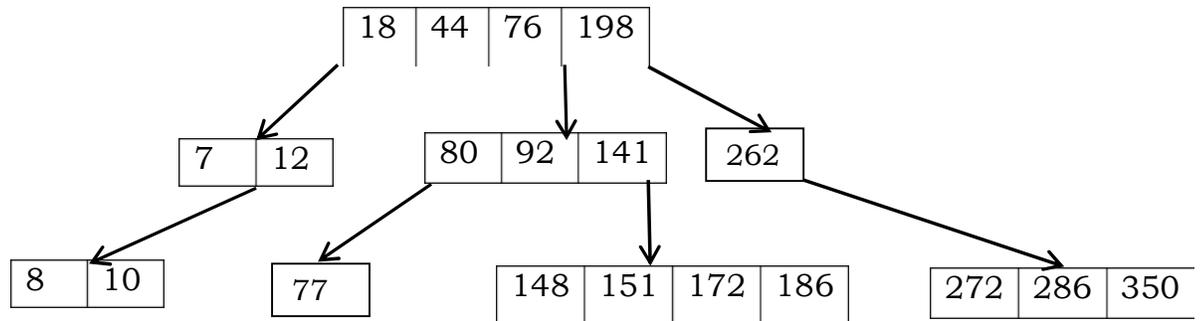


Fig. 3.3.3.1 an M-way search tree (5-way search tree)

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 fig. 3.3.3.1, 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].

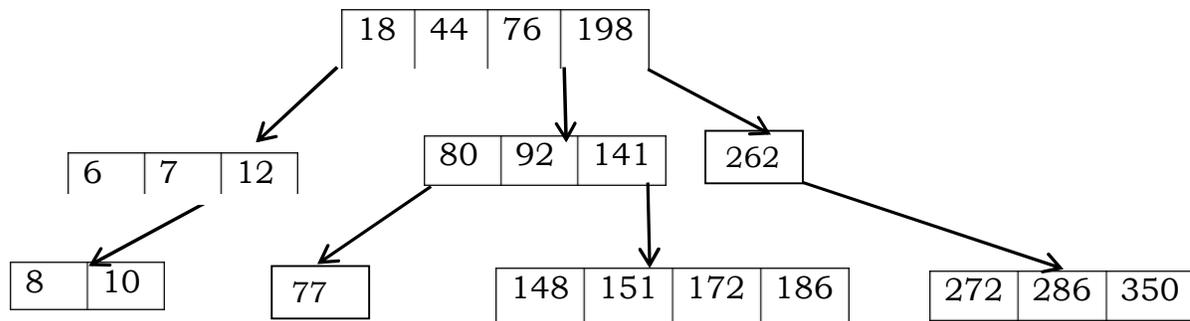


Fig. 3.3.3.2 after inserting 6 into the tree

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 fig. 3.4.3.3 shown below-

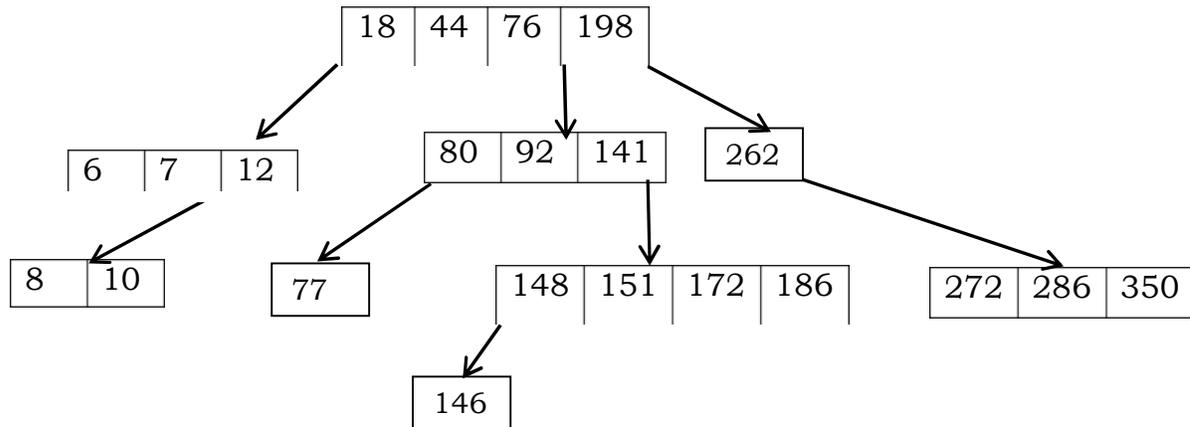


Fig. 3.3.3.3 after inserting 146 into the tree

The coding for insertion in an M-way search tree is given below

```

mTree *insert(int val, 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'
    }
}
    
```

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

```
        return root;
    }
int setval(int val, 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;
}

void fillnode(int val, mTree *c, Mtree *n, int k)
```

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

```
{
    int i;
    for(i=n->num; i>k;i--)
    {
        n->val[i+1]=n->val[i];
        n->child[i+1]=n->child[i];
    }
    n->val[k+1]=val;
    n->child[k+1]=c;
    n->num++;
}

void split (int val, mTree *c, mTree *n, int k,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--;
}

```

3.3.4. DELETION IN MULTIWAY SEARCH TREE

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 ($A_i \neq NULL, A_j = NULL$) 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.
- ($A_i=NULL, A_j \neq NULL$) 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 ($A_i \neq \text{NULL}$ $A_j \neq \text{NULL}$) 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 point.

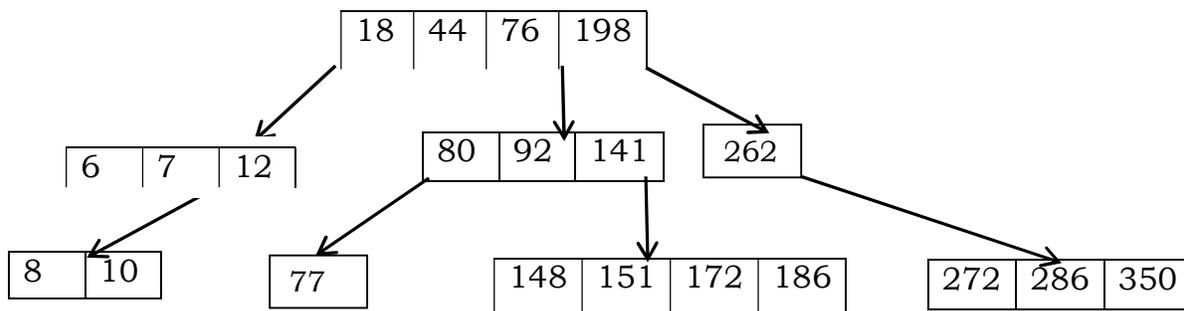


Fig. 3.3.3.4 An m -way search tree

- 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 ($A_i = A_j = \text{NULL}$).
- 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 $A_i = \text{NULL}$ and $A_j \neq \text{NULL}$.
- 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.

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

- To delete 12, we find the node [7, 12] accommodates 12 and the key satisfied ($A_i \neq \text{NULL}$, $A_j = \text{NULL}$).
- 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(int val, 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);
        }
    }
    return root;
}

int delhelp(int val, mTree *root)
{
    int i;
    int flag;
    if(root==NULL)
        return 0;
    else {
        flag = searchnode(val, root, &i);
        if(flag) {
            if(root->child[i-1]) {
```

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

```
        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);
    }
    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)
```

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

```
{
    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) {
            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);
            }
        }
    }
}
```

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

```
void rightshift( 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--;
}

void leftshift(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];
    }
}
```

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

```
        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];
        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.

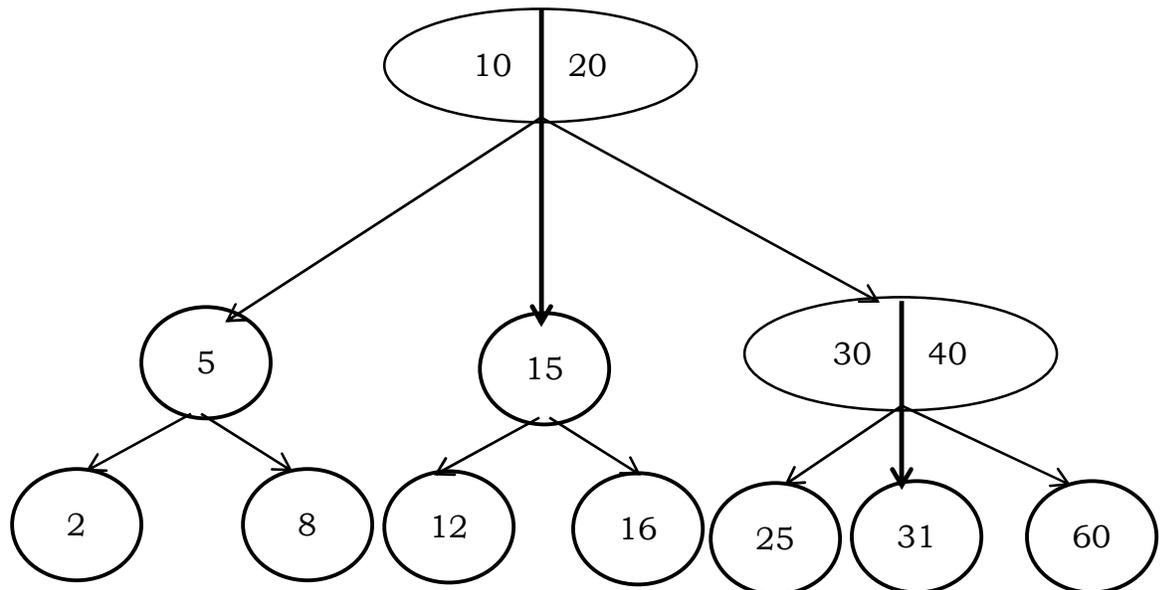


Fig. 3.4.1.1 a 2-3 Tree

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.

3.4.2 SEARCHING OPERATION IN AN 2-3 TREE

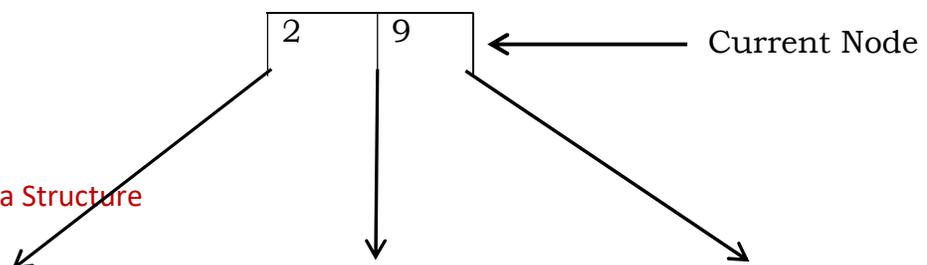
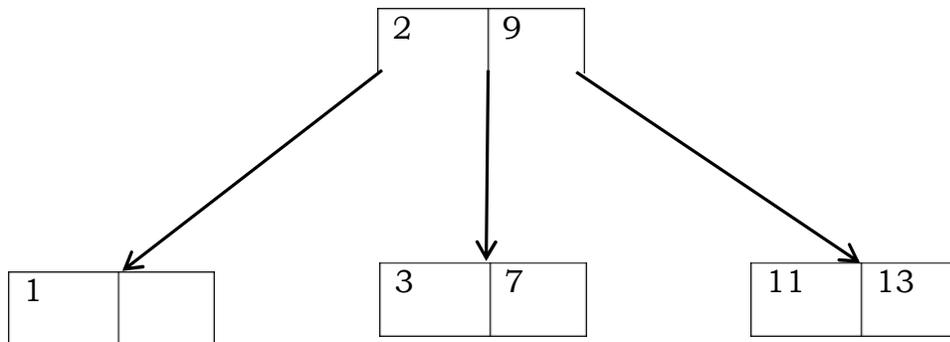
Searching for an item in a 2-3 Tree is same as searching an element in a binary search tree. Since the data elements in each node are treated as ordered, a search function will be directed to be correct sub-tree and eventually to the correct node which consists of the item.

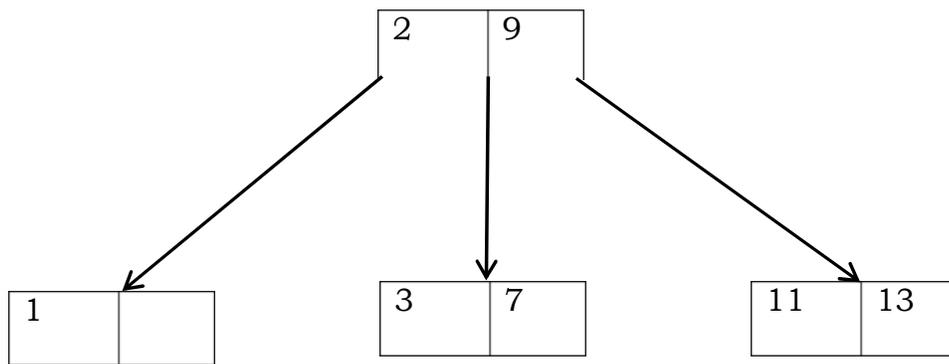
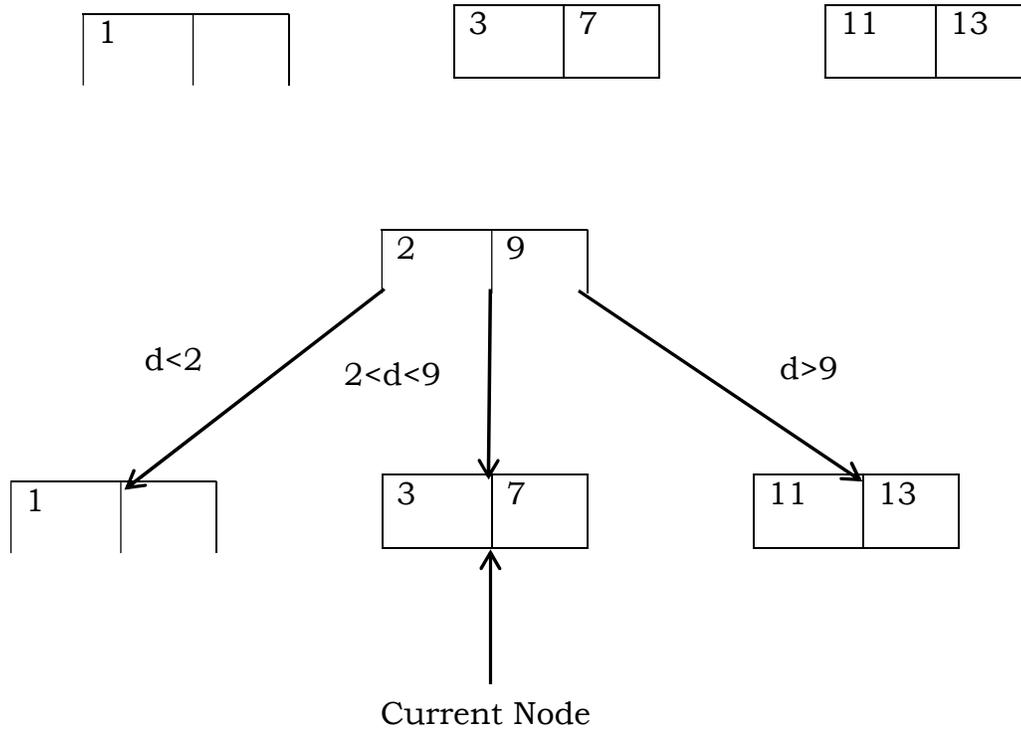
- Let T be a 2-3 Tree and d be the element we want to search. If T is empty, then d is not in T and we will abort the searching.
- Let r be the root of the T .
- Let r is a leaf. If d is not in r , as a result d is not in T . Otherwise, d is in T . In particular, d can be found at a leaf node. We need 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.

Block II (Unit 3: MULTI WAY SEARCH TREES, 2-3 TREES AND SPLAY TREES)

- 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.





5 not found. Return False

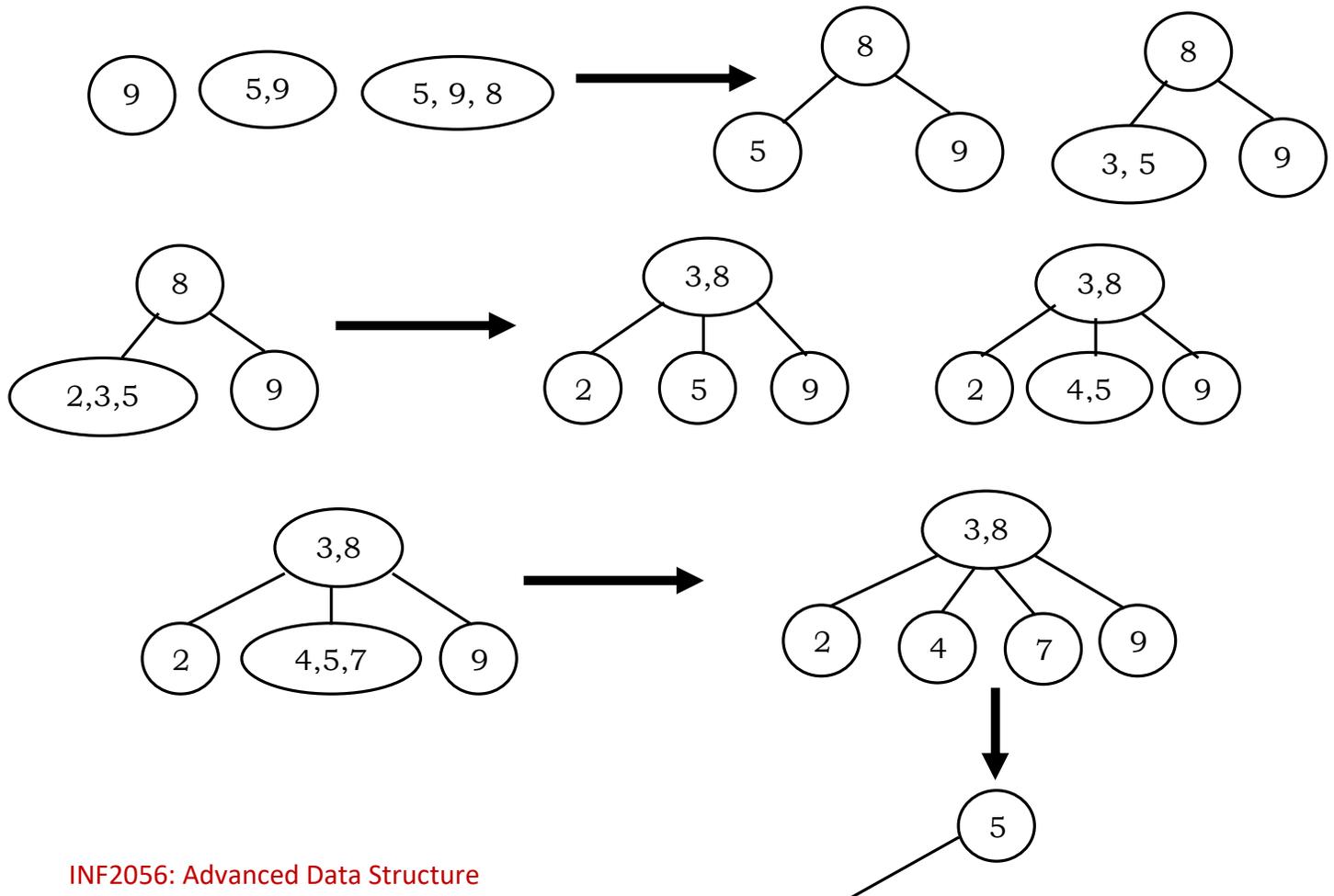
3.4.3 INSERTION OPERATION IN A 2-3 TREE

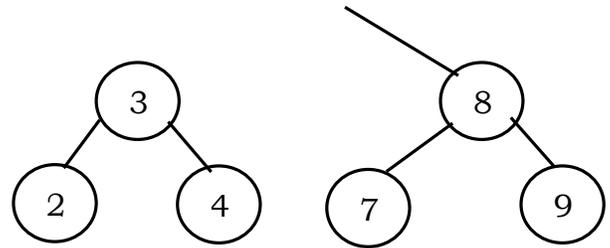
The insert operation takes care of the balanced property of a 2-3 tree. The insertion operation algorithm into a 2-3 tree is quite

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 has 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.

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**

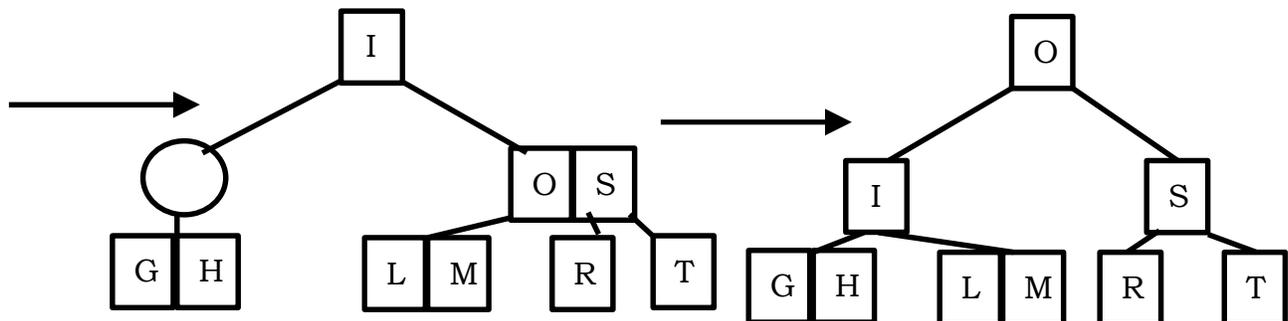
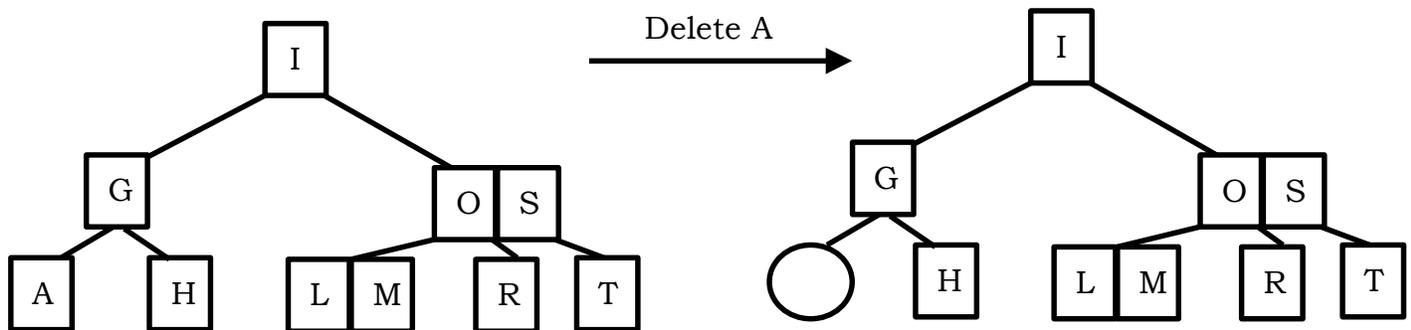
- If n is the root of the tree, then remove the node containing d . Replace the root node with other children.
- If n has a left or right sibling with 3 kids, then
 1. Remove the node containing d .
 2. Use one of the sibling's children.
 3. Fix left value, middle value of n and n 's sibling and ancestors as needed.
- If n 's siblings have only 2 children, then:
 1. remove the node containing **d**

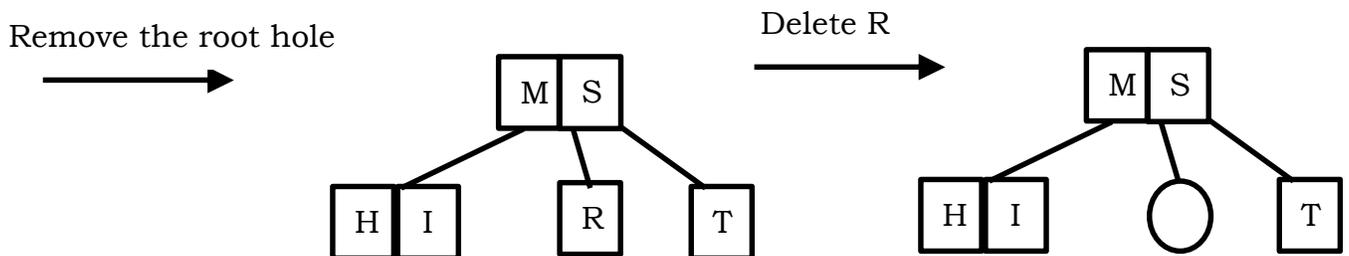
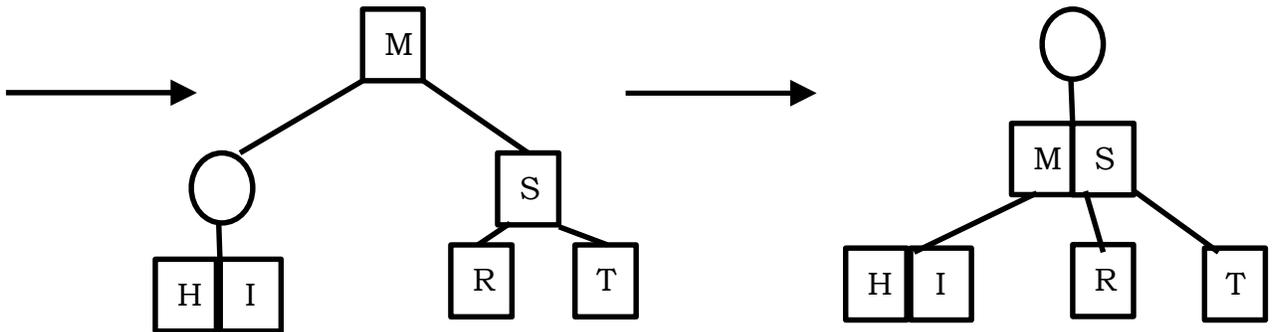
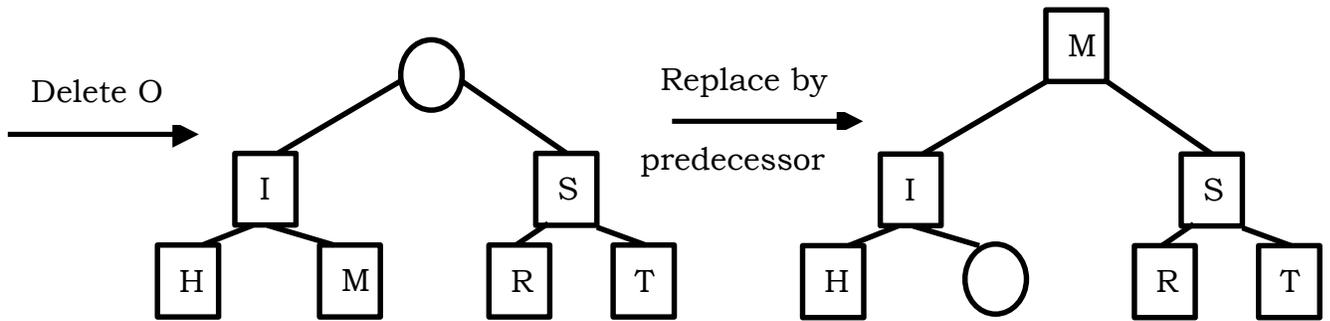
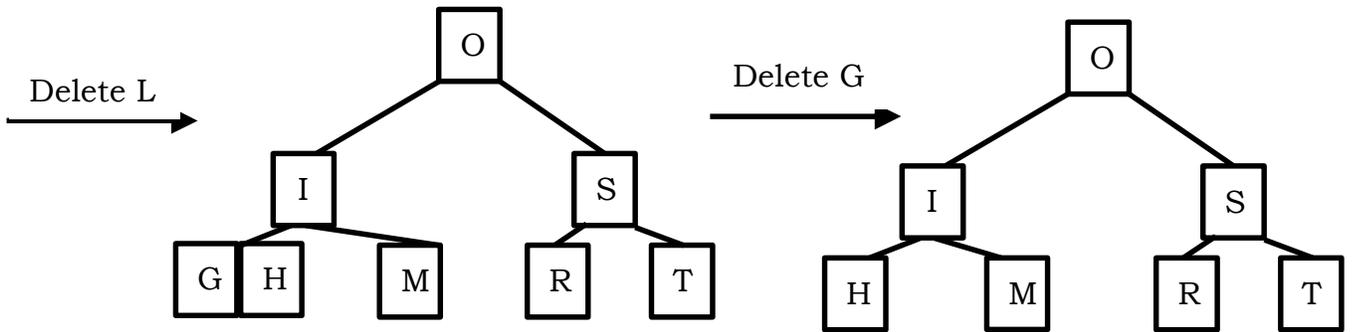
2. make n's remaining child a child of n's sibling
3. fix left value and middle value fields of n's sibling as needed
4. Remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed.

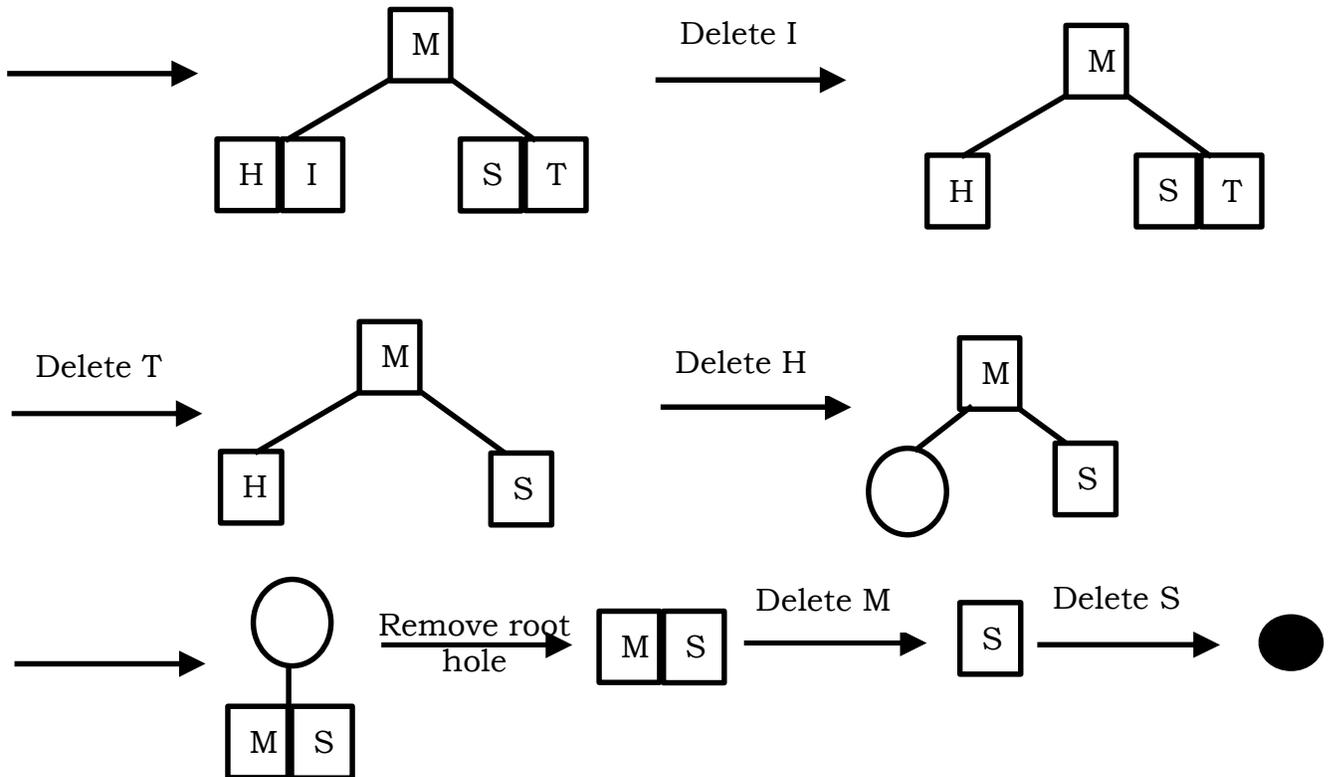
The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing leftMax and middleMax fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).

So the total time is $2 * \text{height-of-tree} = O(\log N)$.

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.







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.

3.5.2 ROTATIONS IN SPLAY TREE

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...

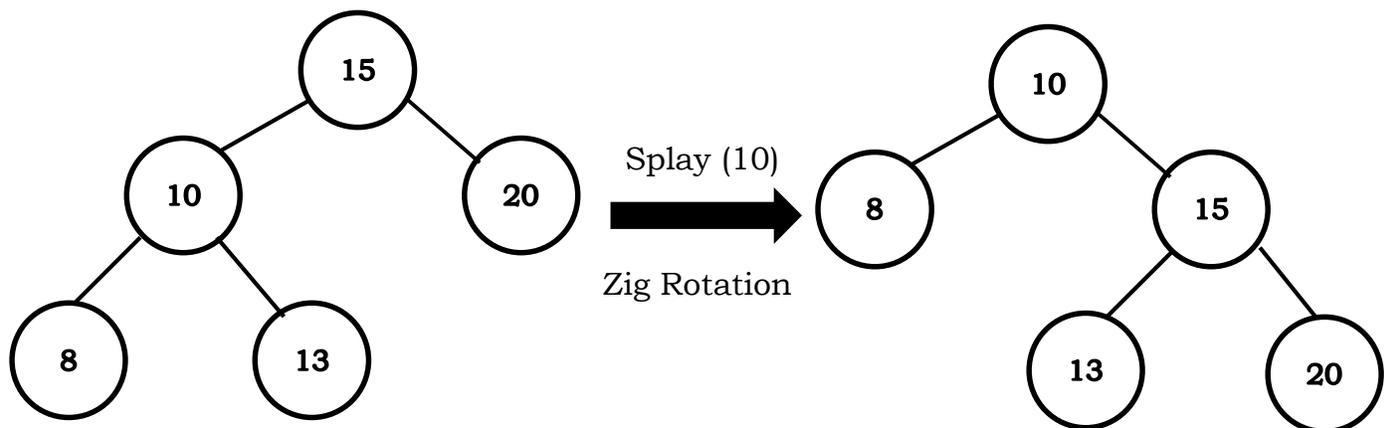


Fig. 3.5.2.1 Zig Rotation

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...

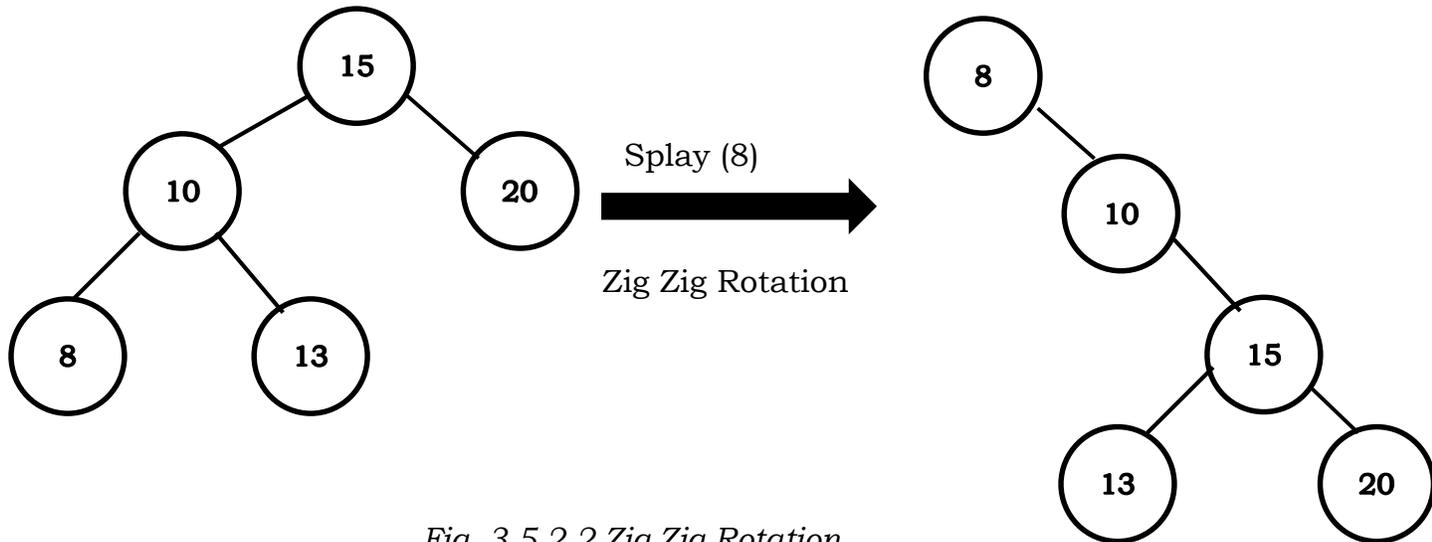


Fig. 3.5.2.2 Zig Zig Rotation

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...

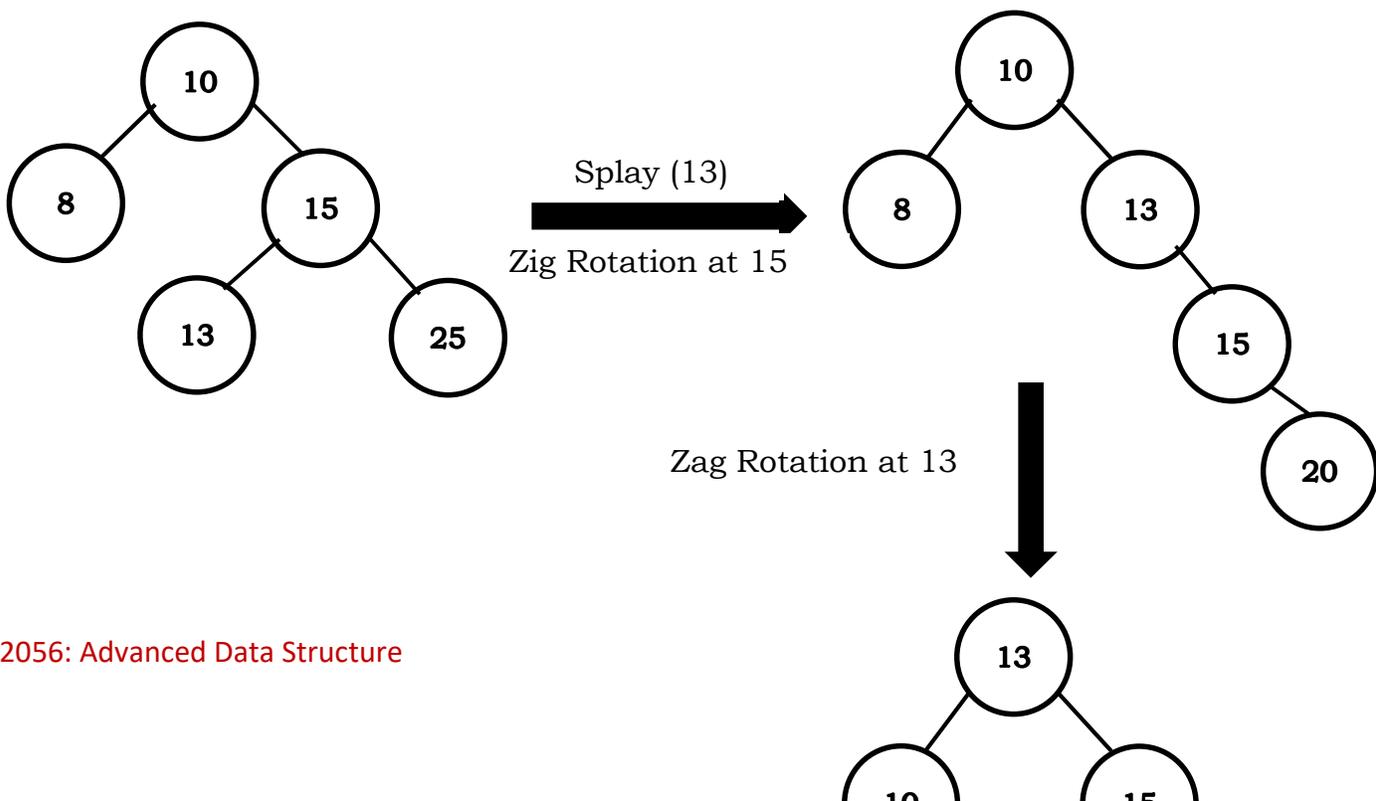




Fig. 3.5.2.3 Zig Zag Rotation

3.5.4 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**

3.5.5 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.

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.

3.7 KEY TERMS

- **Internal Node:** An internal node (also known as an inner node) is any node of a tree that has child nodes.
- **External Node:** an external node (also known as an outer node or leaf node) is any node that does not have child nodes.
- **d-node:** A d-node = a node with d internal children nodes.
- **Order:** The order of a tree refers to the maximum number of children. If a multi way search tree can have maximum m children, then this tree is called as multi way search tree of order m.
- **2-node:** If an internal node has one data element and two children then this node is called 2-node.
- **3-node:** If an internal node has two data elements and three children then this node is called 3-node.
- **Splaying:** When a node is accessed, a splay operation is performed on the node to move it to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves the node 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.

3.8 QUESTIONS AND ANSWERS

Multiple Choice Questions

1. A 2-3 tree is a specific form of _____

- (a) B – tree
 - (b) B+ – tree
 - (c) AVL tree
 - (d) Heap
2. The height of 2-3 tree with n elements is _____
- a) between $(n/2)$ and $(n/3)$
 - b) $(n/6)$
 - c) between (n) and $\log_2(n + 1)$
 - d) between $\log_3(n + 1)$ and $\log_2(n + 1)$
3. 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
4. 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) postorder traversal yields elements in sorted order
 - d) it is B-tree of order 3
5. 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.
6. 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 $O(j \log h)$ time complexity.
 - (c) Sequence of operations with h nodes can take $O(\log h)$ time complexity.
 - (d) Splay trees are unstable trees.
7. 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.
8. Which of the following is an application of Splay Tree?
- (a) Cache Implementation
 - (b) Networks
 - (c) Send values
 - (d) Receive values
9. 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.
10. 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

Answer: 1. (a), 2 (d),3 (b),4 (c), 5 (a), 6 (b), 7 (c), 8 (b), 9 (a), 10 (b).

Fill up the blanks

1. Multi way search trees are not binary search tree because_____.

2. A Multi way search tree has n items. The number of external node for this tree is_____.
3. 2-3 tree is a specific form of _____.
4. For efficient searching of elements we prefer_____ data structure.
5. An internal node in a 2-3 tree is said to be a _____ node if it has two data elements and _____children.
6. The maximum number of children a 2-3 tree can have is_____.
7. Moving a node to the root is called _____ operation.
8. Self-adjusting binary search tree is called _____.
9. Splay trees have _____ complexity.
10. _____ of a Splay tree can be linear when accessing elements in non-descending order.

Answer: 1. it can have more than two children, 2. $n+1$, 3. B-Tree, 4. 2-3 tree, 5. 3 3, 6. 6, 7. Splay, 8. Splay Tree, 9. $O(\log n)$, 10. Height.

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 $N, N-1, N-2, \dots, 4, 3, 2, 1$ 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.

3.9 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 1: PRIORITY QUEUE ADT I

Unit Structure:

- 1.1 Introduction
- 1.2 Unit objectives
- 1.3 Heap
- 1.4 Heap Tree-Based Extended Priority Queue
- 1.5 Min Heap
- 1.6 Max Heap
- 1.7 Summing Up
- 1.8 Answers to Check Your Progress
 - 1.9 Possible Questions
- 1.10 Further Readings

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 TREE

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

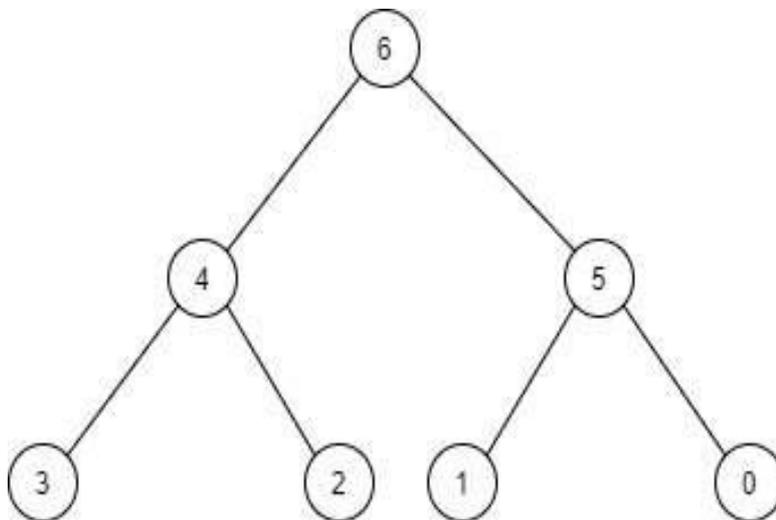


Fig. 1.1 Max heap tree

So, if you perform the traversing based on the above array representation, then Fig.1.1 will represent as a follows

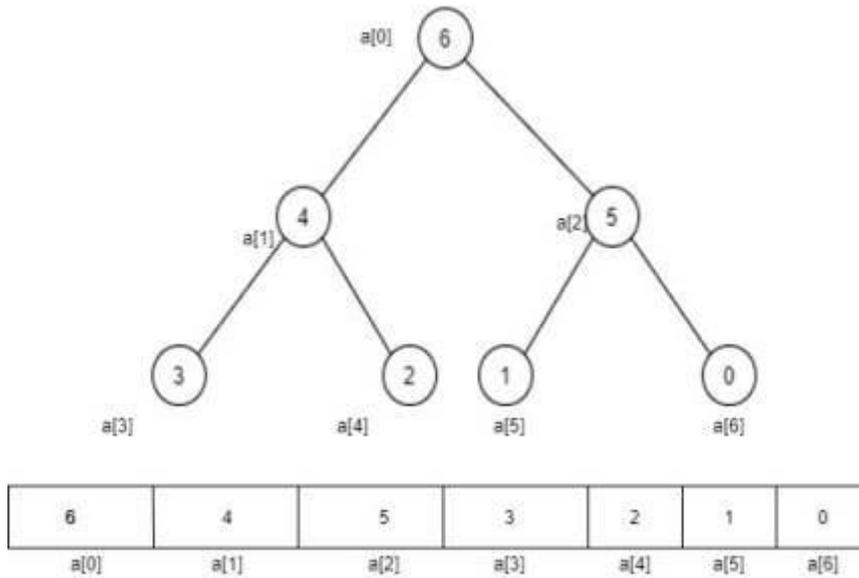


Fig.1.2 array Representation of Heap

Fig1.1 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. Fig.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.

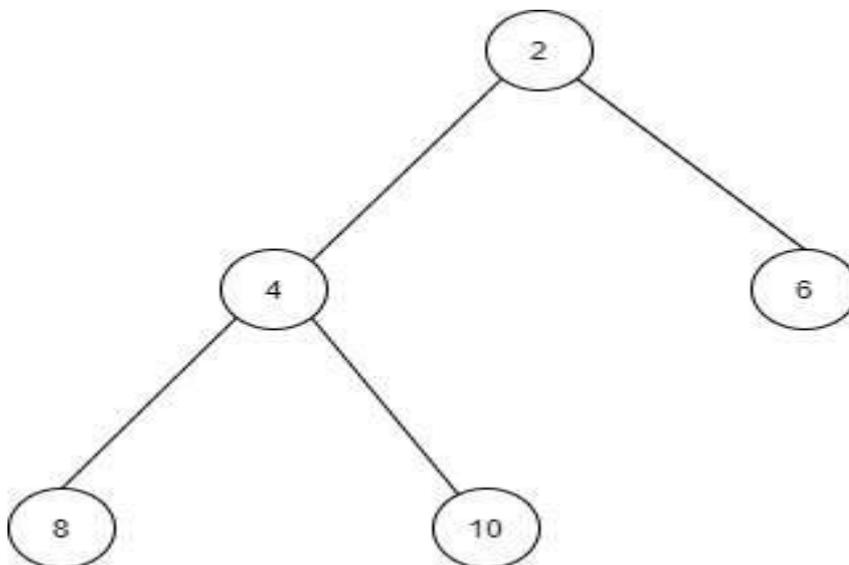


Fig.1.3 Min Heap Tree

1.4 HEAP TREE-BASED EXTENDED PRIORITY QUEUE

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 takes $O(n)$ time. In heap tree implementation, the complexity is $O(1)$ and $O(\log n)$. 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 3, 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, then the 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).

```
int Extract_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.

```
void Increase_Value (int a[ ], int i, int val)  
{  
    if(val<a[ i ])  
    {
```

```
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).

```
void insert_value (int a[ ], int val)
{
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 $O(\log n)$.

1.5 MIN HEAP TREE

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. The following is an example of a min-heap tree.

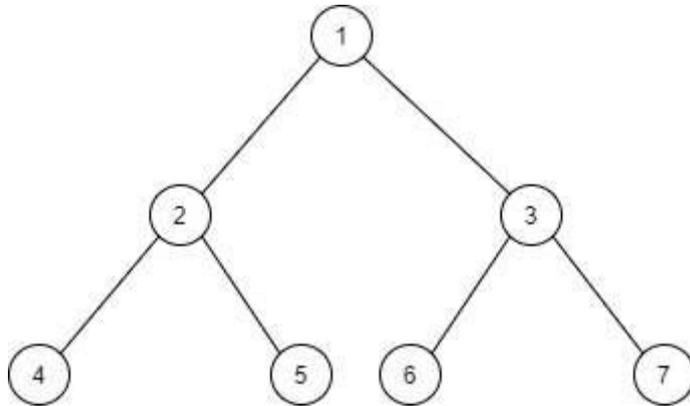


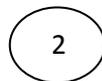
Fig.1.4 Min Heap Tree

In the fig. 1.4, the elements {1, 2, 3, 4, 5, 6, 7} are inserted as follows.

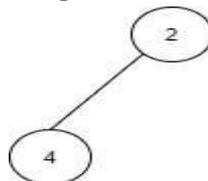
- i) The first element is 1, as the heap is null, you can directly insert 1.
- ii) The second element is 2 and it is placed in the second position of the tree as per array position.
- iii) The third element is 3, and it is more than 1 and 2. So no need to perform heapifies.
- iv) Then you can insert 4, 5, 6, and 7 by following the steps i to iii.

Let's take another example. The array is $a = \{2, 4, 1, 6, 3\}$

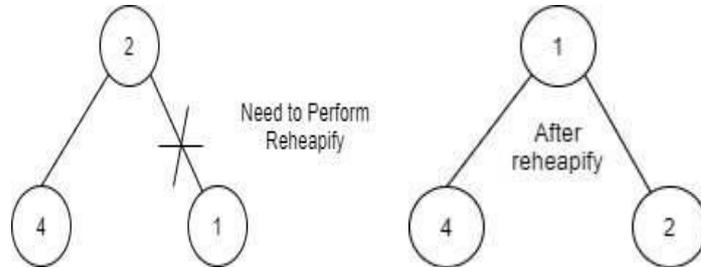
- i) At first, insert the first element in the min-heap.



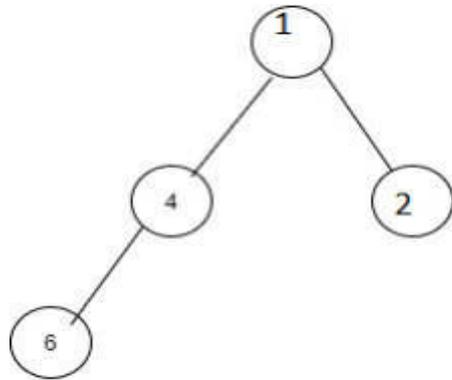
- ii) Now, you can insert 4. As 4 is more than 2, 4 will be a child of 2 and it will be placed left of 2.



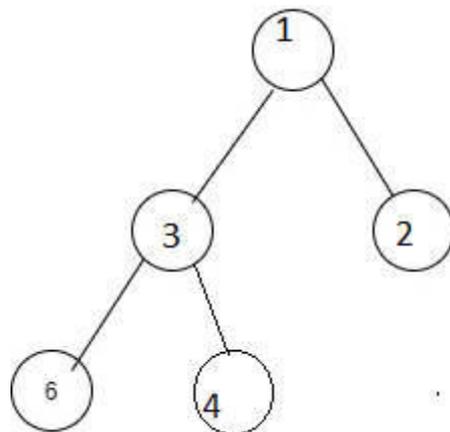
- iii) The next element is 1. If you insert 1 on the right of 2, it will violate the property of min-heap. So, you need to heapify it. After heapify, the 1 will go to place 2 and 2 will come to the place of 1.



- 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 $O(1)$.
- ii) The complexity of heapify is $O(h)$
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.

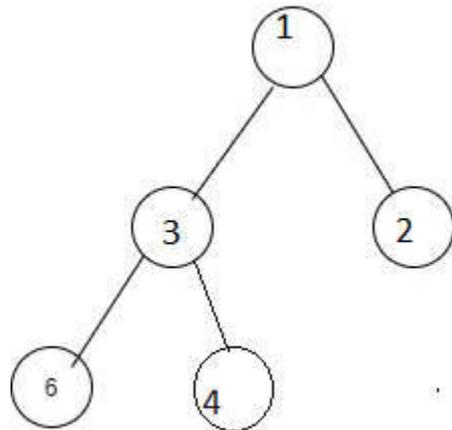


Fig.1.5 Min Heap Tree

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.

- 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 $O(\log n)$.

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.

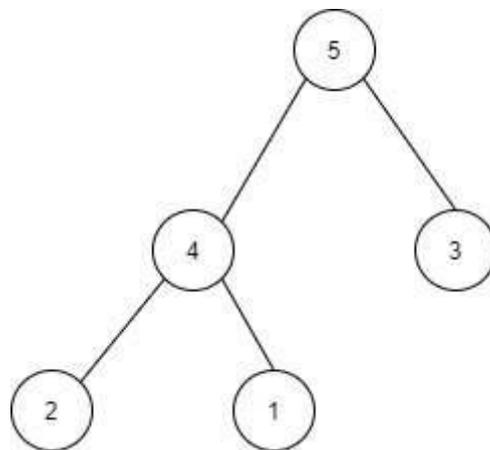


Fig.1.6 Max Heap Tree

In Fig.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.

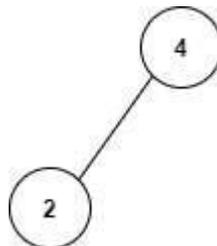
- iii) The next node is 3. So, 3 will go to the right of 5 by maintaining the max heap tree property.
- iv) Node 2 will go to the left of 4 by maintaining the max heap tree.
- v) The last node 1 will be placed on the right of 4.

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.

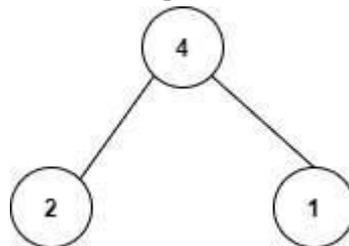
- i) First, insert the 2 in the empty tree.



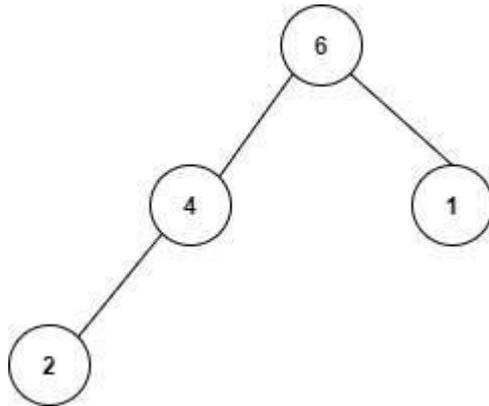
- ii) In this step, element 4 is added to the tree. And it will go to the left of 2. It has violated the property of the max heap. So, you need to perform the heapify. After heapify, node 4 is in the root position and 2 is placed as a left of 4.



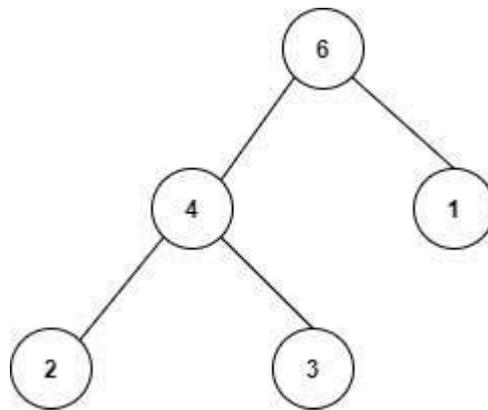
- iii) The next element 1 is placed as a right of 4. So, no issue and no need to perform the heapify.



- 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.



- 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.

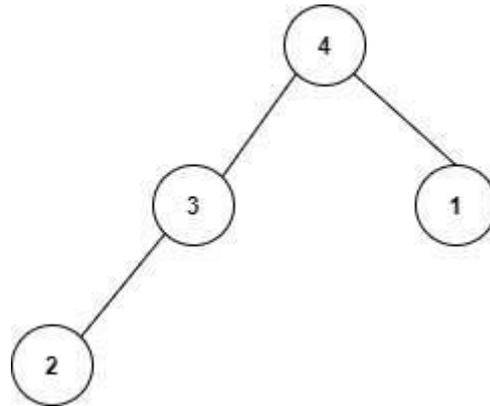


After insertion, the following steps are used for the deletion in the max heap tree.

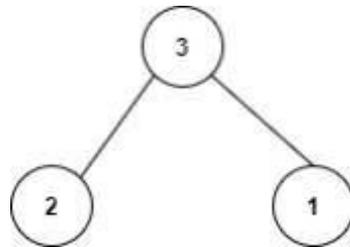
- i) Delete the root node (greatest priority).
- ii) Then, the last node according to the position of the tree will be placed as a root node.
- iii) After interchanging, its max heap tree property is not maintained, perform the heapify.
- iv) Follow the above steps.

So, consider the above three and perform the deletion operation in the tree.

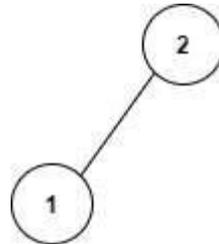
- i) The first deleted node is 6. After deletion, 3 will be placed in the position of 6. But it violates the rule of max heap. So, you need to perform the heapify. After heapify, 4 will act as root, and 3 will be as left of 4.



- 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, delete node 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 heap are in the following steps.

- i) If you add a node at a level of height h , then the complexity of adding is $O(1)$.
- ii) The complexity of reheapify is $O(h)$
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 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 tree?
- 5. What is a min heap tree?
- 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 $A[\text{Parent}(i)] \leq A[i]$
 - v) The procedure FindMin() to find the minimum element and the procedure DeleteMin() to delete the minimum element in min heap take logarithmic time.

1.7 SUMMING UP

- 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 $O(\log n)$.

1.8 ANSWER TO CHECK YOUR PROGRESS

1. 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.
2. If a heap node contains two children, then the heap tree is a binary heap tree.
3. i) True ii) True iii) False iv) 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. $O(\log n)$
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 heapify?
- 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.

1.10 FURTHER READINGS

Data Structures Using C by Reema Theraja Publisher: Oxford Publication

UNIT 3: Partition ADT

Unit Structure:

- 3.1 Introduction
- 3.2 Unit objectives
- 3.3 Union Find Data Structure
- 3.4 Disjoints Sets and Union-Find
- 3.5 Union Find Rank and Path Compression
- 3.6 Summing Up
- 3.7 Answers to Check Your Progress
- 3.8 Possible Questions
- 3.9 Further Readings

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 disjoint 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 disjoint 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 disjoint 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 $\text{Find}(2) = A$. It means that the value 2 is present in set A. Again, $\text{Find}(6) = B$. It means that 6 are present in set B. If $\text{Find}(A) = \text{Find}(B)$, then it is considered that the elements are present in the same set.

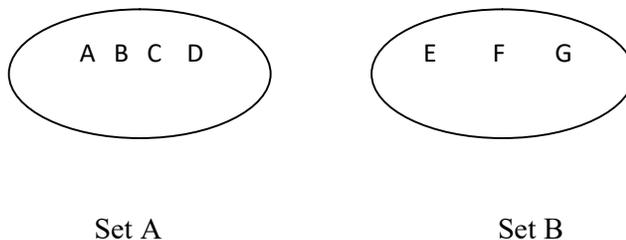
For Union operation, the two sets are joining together. $UNION(A, B) = \{1, 2, 3, 4, 5, 6, 7, 8\}$. These two operations of the data structure are used to detect the cycle in an undirected graph.

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. Let's say 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 \emptyset . 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 determines which subset a particular element is in.
- ii) Union:* It is used to find the union of two sets.

For better understand, let's say you have two sets.



In the above, two sets are there. Set A has 4 elements i.e., A, B, C, and D. Set B has 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 an undirected graph contains a cycle or not. For this, Union-Find Algorithm is used using an array representation. 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);
}
```

Let's have a graph of five vertices as shown in the diagram.

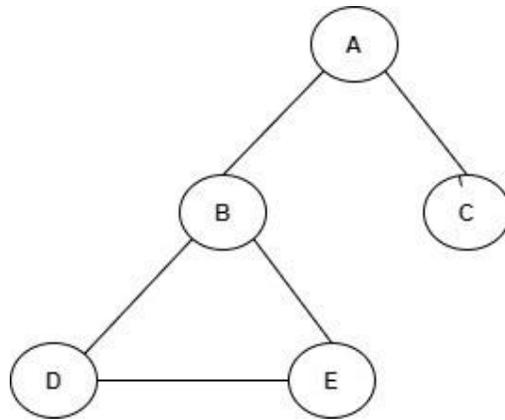


Fig. 3.1. A graph with 5 vertices.

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

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., $\text{Find}(A) = A$. It means that the parent of A is A. Again the value of B is -1, i.e., $\text{Find}(B) = B$. 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 $\text{Find}(A)=B$, and $\text{Find}(C)=C$. As they belong to two different sets, so union operation is required to perform, i.e., $\text{Union}(A, C)$. After union, the parent of B is C.

B	C	-1	-1	-1
A	B	C	D	E

- iii) The next edge is BD. $\text{Find}(B) = C$, $\text{Find}(D)=D$. So do union operations using $\text{union}(BD)$. After union the parent of C is D.

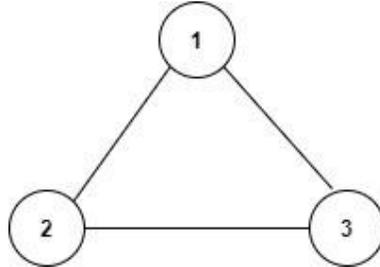
B	C	D	-1	-1
A	B	C	D	E

- iv) Take the edge BE. $\text{Find}(B)= C$ and $\text{Find}(E)=E$. Do union using $\text{union}(B,E)$. The parent of D is E.

B	C	D	E	-1
A	B	C	D	E

- v) Now take the edge DE, $\text{Find}(D) = E$, $\text{Find}(E)=E$. It means that the two values are the same. So there is a cycle.

Lets You have an example. Consider the following undirected graph. The contains 3 vertices as follows.



For the Find-Union Algorithm, consider the following array.

1	2	3
-1	-1	-1

Initially, the value parent of each node is -1. It means that the parent of each node is itself. Now apply the following steps.

- i) Consider the dge 1-2. $\text{Find}(1) = 1$, $\text{Find}(2) = 2$, They are not same. So, do Union using $\text{Union}(1,2)$. After performing union the parentent of 1 will be 2.

1	2	3
2	-1	-1

- ii) Now Consider the next age 2-3. $\text{Find}(2) = 2$, $\text{Find}(3) = 3$. Do apply $\text{Union}(2,3)$ 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. $\text{Find}(1)=2$ and $\text{Find}(3)=3$. It means that they are in the same set. So there is a cycle in the undirected graph.

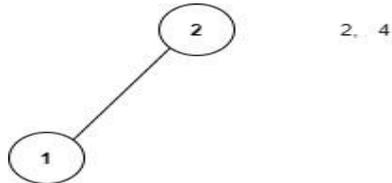
Check Your Progress-I

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 represent itself as a parent in Union Find.
 - ii) The Worst case complexity of Union find is $O(n)$.

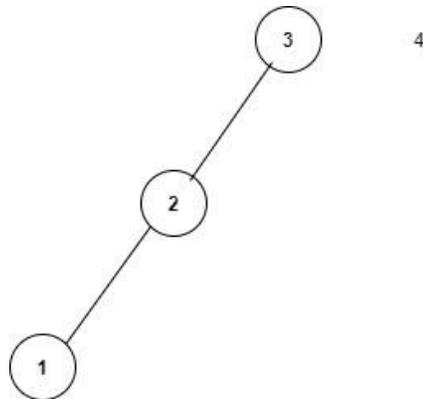
3.5 UNION BY RANK AND PATH COMPRESSION

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. Lets understand by considering an array $A = \{1,2,3,4\}$

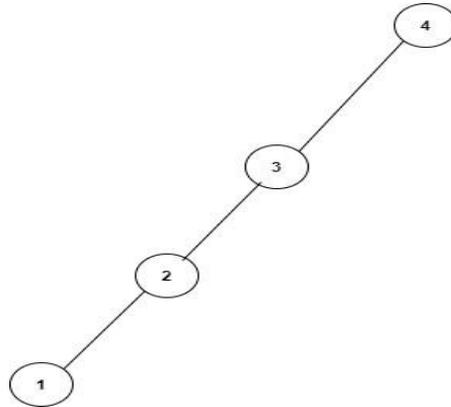
- i) Initially, all the elements are in a single element set.
- ii) Do perform union in between 1 and 2 as follows.



- iii) Now perform unions 2 and 3 as follows.

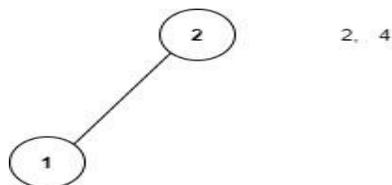


- iv) Now perform union in between 3 and 4 as follows.

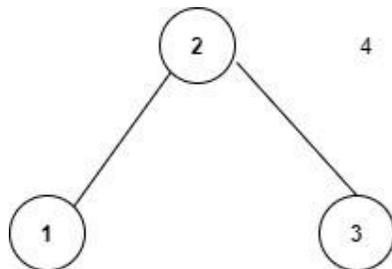


The above operation takes $O(n)$ times but it can be optimized to $O(\log n)$ using the union rank algorithm. The union by rank always attached a smaller depth tree under the root of the deeper tree. In this technique, instead of finding the height, path compression is used and then the rank is not equal to height. The size of the tree can also be used as a rank. Let understand the union rank for the above example.

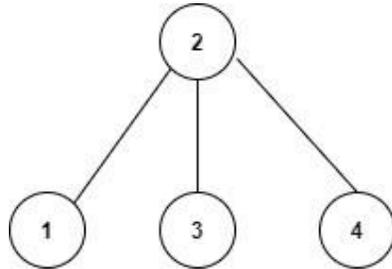
- i) Initially, all the elements are in a single element set.
 ii) Do perform union in between 1 and 2 as follows.



- iii) Now perform unions 2 and 3 as follows.



- 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.

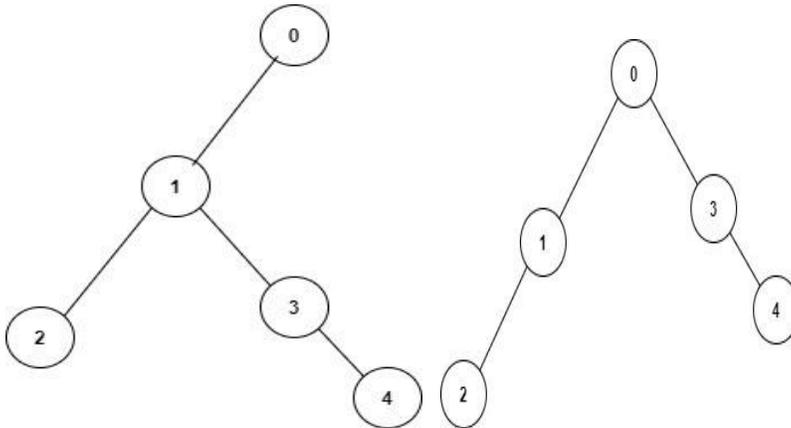


Fig. 3.2. a) Before Path Compression b) After Path Compression

The pseudocode for union by rank and path compression are given below.

A. UnionbyRank (m, n)

- i. mRoot = Find(m)
- ii. y=nRoot = Find(n)
- iii. // if m and n are already in the same set
 - 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 disjoint 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.

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 pseudocode for union by rank is given below.

UnionbyRank (m, n)

- v. mRoot = Find(m)
- vi. y=nRoot = Find(n)
- vii. // if m and n are already in the same set
 - a. if mRoot == nRoot
 - b. return
- viii. // 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. $mRoot.rank := nRoot.rank + 1$

10. The pseudocode for union path compression is given below

FindbyPathCompression(m)

- i. if $m.parent \neq m$
- ii. $m.parent := Find(m.parent)$
- b. return $m.parent$

3.7 ANSWER 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 behavior 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 $O(\log n)$ 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 $O(\log n)$ time which is less than the traditional Union Find Algorithm.

7. The worst-case time complexity of Union Rank and Union Path Compression is $O(\log n)$.

3.8 POSSIBLE QUESTIONS

Short answer type questions:

- i) What are disjoint sets? Give example.
- ii) Can you merge a disjoint set in $O(\log n)$ 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-PathCompression with their pseudocode.

6.13 FURTHER READINGS

1. <https://www.geeksforgeeks.org/disjoint-set-data-structures/>
2. <https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/>

UNIT 4: B Tree and B+ Tree

Unit Structure:

- 4.1 Introduction
- 4.2 Unit objectives
- 4.3 B Tree
- 4.4 Operations of B Tree
- 4.5 B+Tree
- 4.6 B+ Tree Operations
- 4.7 External Sorting
- 4.8 Complexities of B tree and B+ tree
- 4.9 Summing Up
- 4.10 Answer to check your progress
- 4.11 Possible Questions
- 4.12 Further Readings

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

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 $\lceil m/2 \rceil$ children.
- Every node except the root of a B tree contains at least $\lceil m/2 - 1 \rceil$ 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 $\lceil m/2 \rceil$ children whereas the leaf has 0 children.
- All leaf nodes must be at the same level.

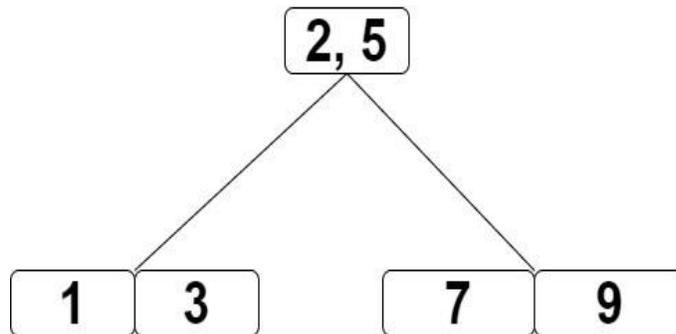


Fig.4.1. Example of B Tree.

4.4 OPERATIONS OF B TREE

A B-tree has the following operations.

- i) Search operation

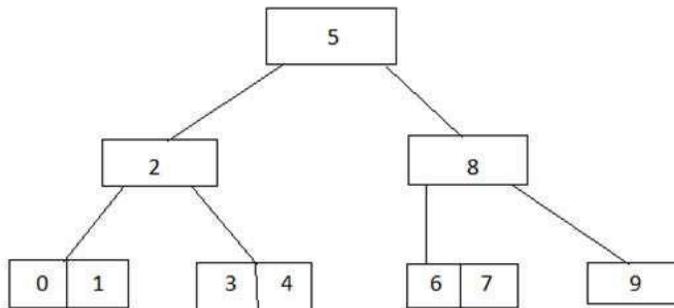
- ii) Insert Operation
- iii) Delete Operation

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.
- i) 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 5,9,3,7,1,2.

Let's consider that the tree is empty.

- i) As the tree is empty, insert the 5.



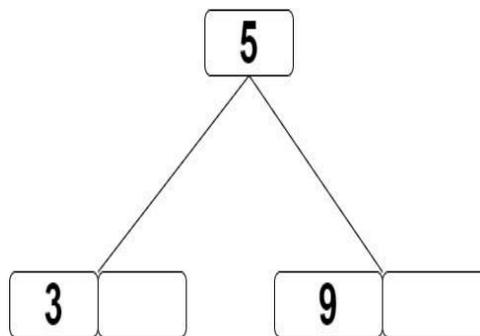
- ii) The root contains a maximum of 2 elements like the order of the tree is 3. So, elements 9 also can be inserted in the root node. But, insertion should take place in a sorted manner. As 9 is greater than the 5, so it will be right of 5 in the same node.



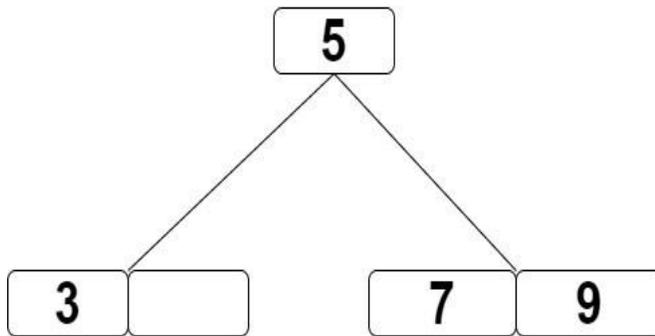
- iii) The next element is 3. But the first node is full, so insertion is possible in the node. Because 2 nodes are possible to insert in any node. So the element 3 will be added to the next level after splitting the first node. Before splitting, lets 3 is inserted in the first node which is not possible. So, split it by finding the median element. In that case, 5 will go up and acting as the root of 3 and 9.



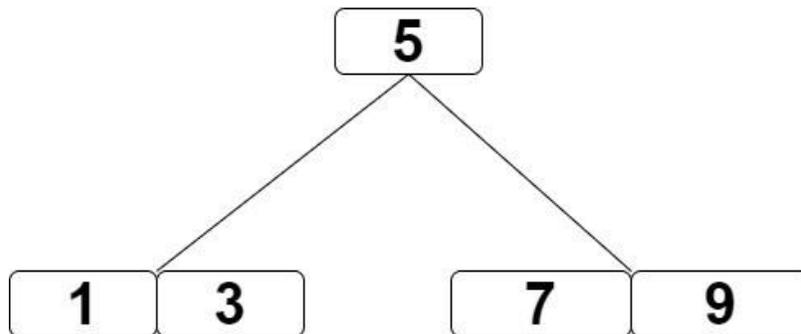
So the elements are rearranged as follows.



- 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 can not be the right of 9. So, after rearranging the tree will be as follows.



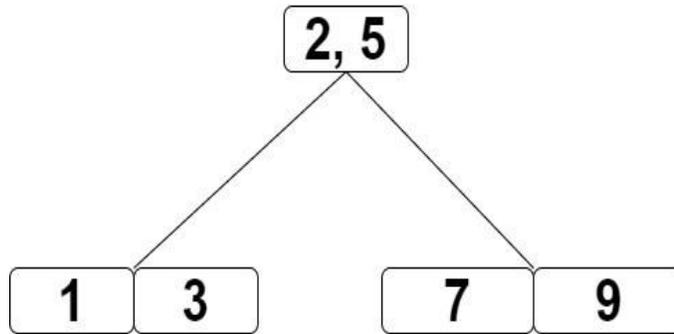
- v) The next element is 1. Again compare 1 and root key. The element is lesser than 5, so it will be placed at the left of 5. After changing the positions of 1 and 3 for sorting, the tree is as follows.



- vi) The next element is 2. Again compare with root 5. It is lesser than root, so move to the left of root. But the left node of the root is already filled up. So, splitting is 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.

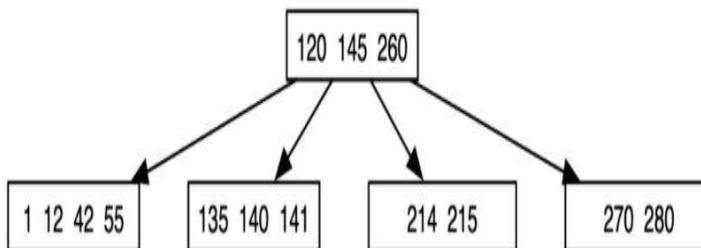
- i) The target may be present in the leaf node
- ii) The target may be present in the internal node.

Now consider the case [i], i.e., the target is present in the leaf node.

Again there are two possibilities.

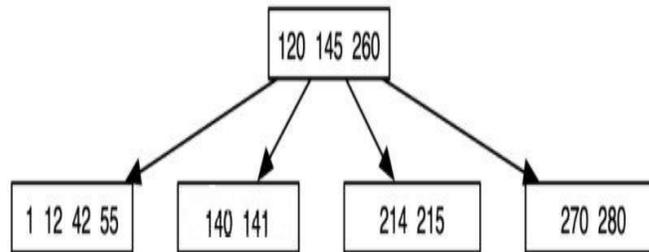
- a) The leaf node may have more than a minimum number of keys.
- b) The leaf node contains the minimum number of keys.

For better understand, consider the following B Tree of order 5.

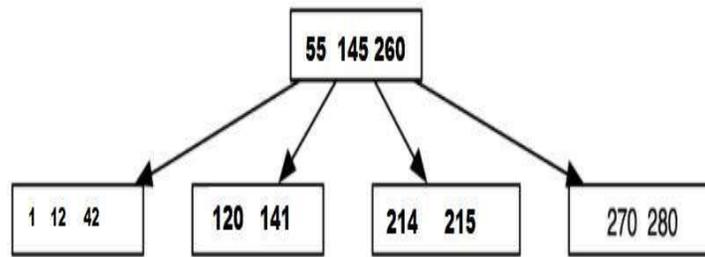


In the above tree, the minimum number of keys in the tree is $\lceil m/2 - 1 \rceil = 2$, and the maximum number of keys in the tree is $\lceil m - 1 \rceil = 4$. So, let's consider the first possibility of a leaf node and the target element is 135 and

- 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.

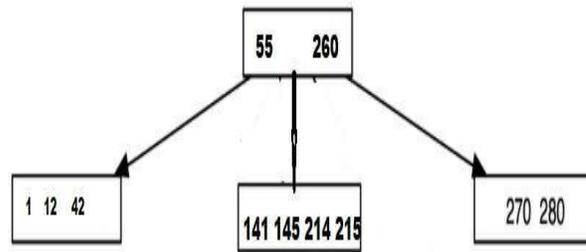


- ii) In the above B tree, all the node contains minimum 2 keys. It means that it follows the properties of the B Tree. Now delete node 140. If you delete node 140. It will violate the rule of the B tree because that node will contain only one key. In this case, the node will borrow a key from the sibling 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.



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.

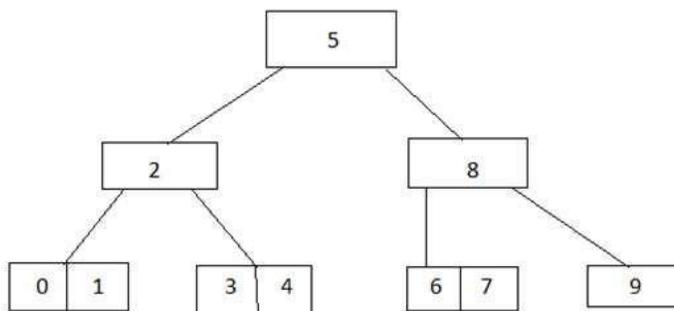
Now delete the key 120 from the B tree. If key 120 will be deleted directly, it will violate the properties of the B tree. In this situation, merging can be done to validate the B tree. But the merging will happen through the root key of the 1st and 2nd leaf node of the tree, i.e., 55. By consider the 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 jong 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.

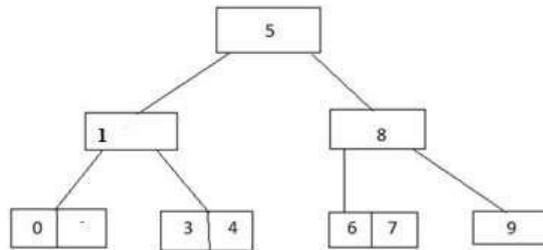
- i) The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.
- ii) The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.
- iii) If either child 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.

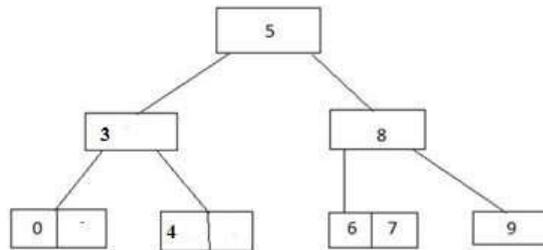


In this case of deleting key 2 from the internal node, the left child of the internal node has more than the minimum number of keys. So, it will follow the rule [i] as mentioned above. After deleting key

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 inorder successor of 1, i.e., 3.



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: $h(\min) = \lceil \log(m + 1) \rceil - 1$
- 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: $h(\max) = \lceil \log_t(n + 1)/2 \rceil$ where $t = m/2$.

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 multiway 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. The root node has 4 keys. How many leaf nodes are there in the tree?

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 can no 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 at least 2 children.
- iii) Each node except the root node has at least $m/2$ 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 $\lceil m/2-1 \rceil$ keys.

4.6 OPERATIONS OF B++ TREE

Like the B tree, the following operations are also present in the B+ Tree.

- i) B+ Tree insertion
- ii) B+ 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 = $\lceil m/2 - 1 \rceil = 1$
 - d) The maximum number of key in a node = $m - 1 = 3$.
- ii) Take the first element 1 and insert the first node where you can insert maximum of 3 keys.

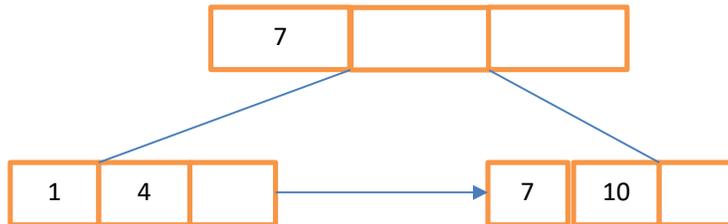


- iii) The second and third elements (4,7) are also inserted in the first node as below.

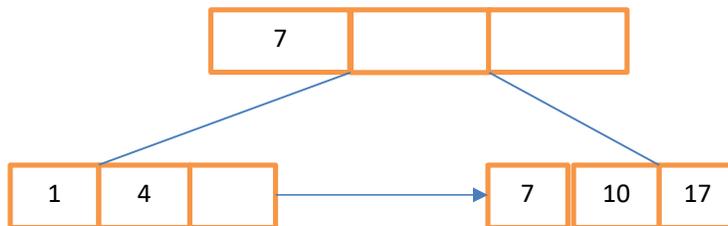


- iv) Now to insert key 10, a node will be required as 10 can not be inserted in the first node because it is already filled up. So, splitting will be required in this situation. On splitting 7 will go up and 1,4 will be present in the

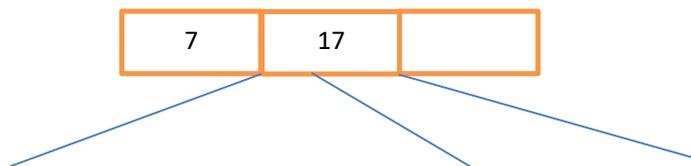
left leaf node and 10 will be present in the right leaf node with node 7. Node 7 will be present in the root node as well as in the leaf node because it is a B+ tree. In the B+ tree, each node should be present in the leaf node. The leaf node is connected by a link as follows.



- 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].

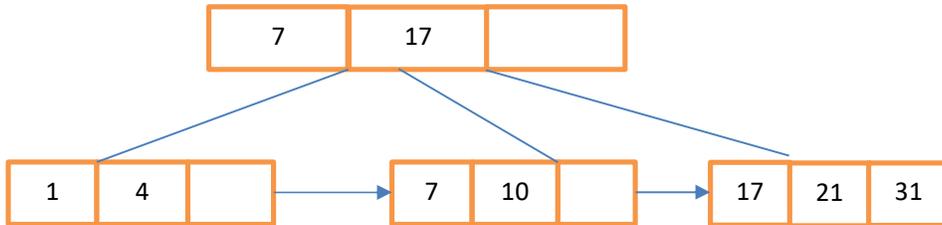


- vi) The next key is 21. It will come to the right of 17 that is not possible as it is already filled up. So, splitting is required. After splitting the element 17 will go up and [7, 10] will be in left and [17, 21] will be the right leaf node. Here also 17, will present in the leaf node though it present in the root as an index.

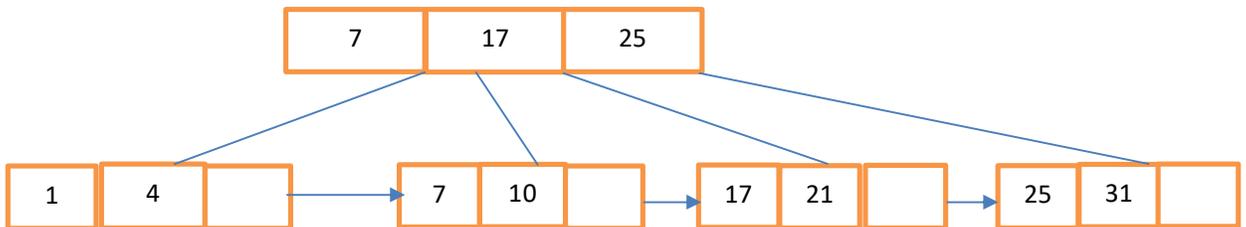




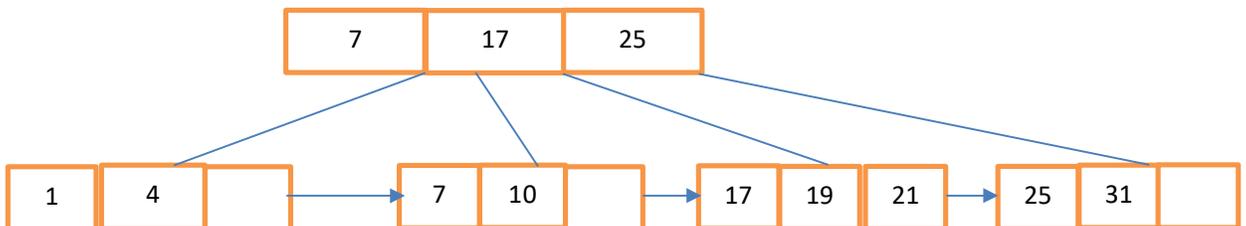
vii) Next is 31. The 31 will be added as a right of 21.



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.

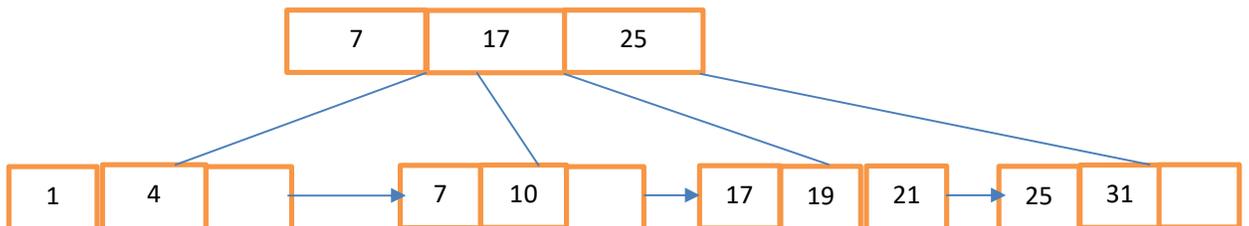


The deletion operation of the B + tree is the same as with the B tree. Let's discuss the deletion of the B+ tree with the following example. Let, the key to be deleted from the B+ tree is present only at the leaf node, not in internal nodes. There are two cases for it:

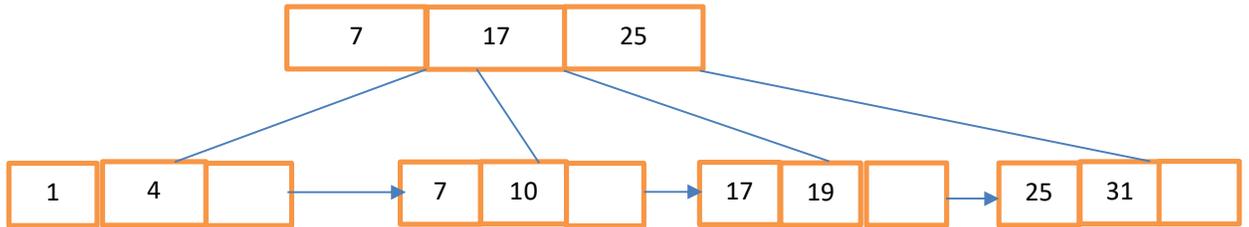
- i) If there is more than the minimum number of keys in the leaf node then simply delete the key.
- ii) If there is an exact minimum number of keys in the node then delete the key and borrow a key from the left or right sibling. Add the median key of the sibling node to the parent.

Lets, 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.

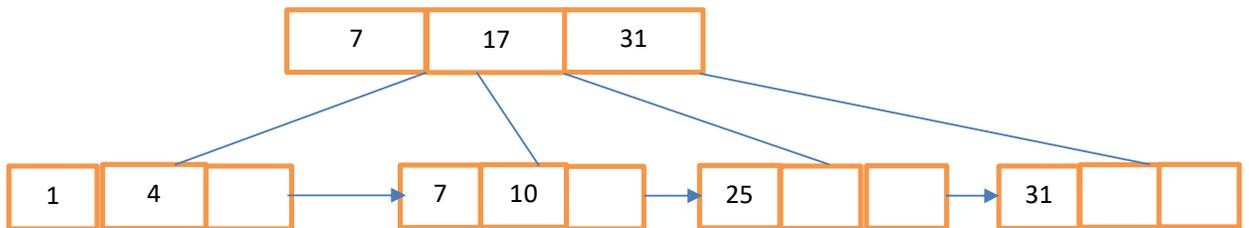
- e) The maximum children = $m = 4$
- f) The minimum children = $m/2 = 2$
- g) The minimum number of key in the tree = $\lceil m/2 - 1 \rceil = 1$
- h) The maximum number of key in a node = $m-1 = 3$.



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.



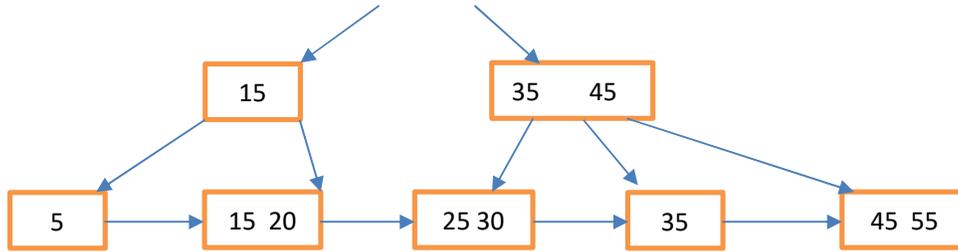
Now again delete 17 and 19. First, delete 19. As the nodes have two keys, so simply delete node 19. After deleting 19, the node contains only 17, i.e., case ii of condition 1. So, delete node 17, but borrow a key from the right sibling. After borrowing, 31 will go up and 25 will come as left of 31 in the left leaf node as follows.



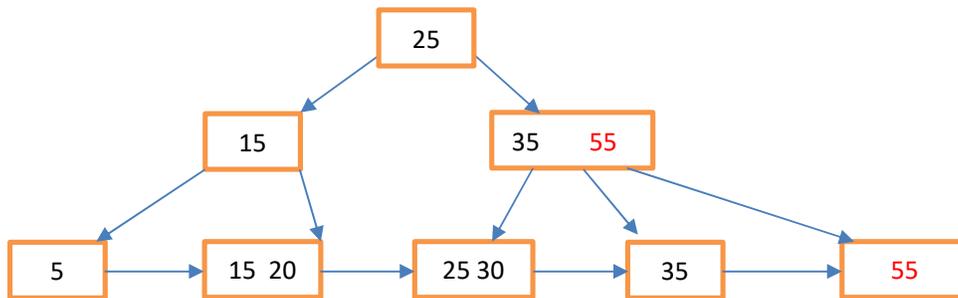
Sometimes the key to be deleted is present in the internal nodes as well. Then you have to remove them from the internal nodes as well. There are the following cases for this situation.

- i) 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 inorder successor.



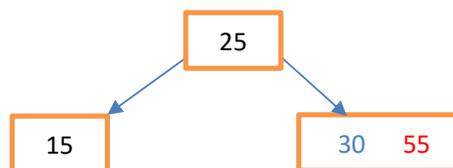


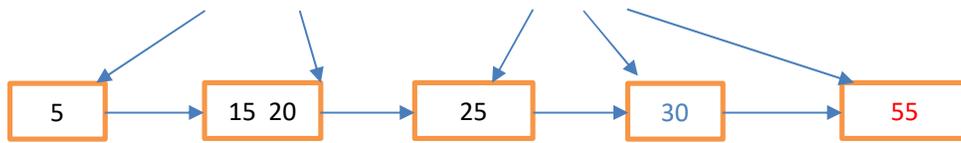
The above figure, lets you want to delete key 45. The node that contains 45, has more than the minimum number of keys. So simply delete the key 45, and replace it with an inorder successor value i.e., 55.



- 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 sortings is merge sort where a small chunk of elements are placed in a RAM and then sort them. Finally 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	$O(t \log n)$

Sr. No.	Operation	Algorithms	Time Complexity
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 Flase)

4.9 SUMMING UP

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 $\lceil m/2 \rceil$ children.
 - Every node except the root of a B tree contains at least $\lceil m/2 - 1 \rceil$ 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 $\lceil m/2 \rceil$ 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 at least 2 children.
 - Each node except the root node has at least $m/2$ 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 $\lceil m/2 - 1 \rceil$ 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 Complexities of the B tree is $O(\log n)$ and the B+ tree are $O(t \log t n)$, $O(\log n)$ $O(M \cdot \log n + \log L)$, and $O(\log n)$ $O(M \cdot \log n + \log L)$.

4.10 ANSWER TO CHECK YOUR PROGRESS

3. 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.
4. 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 $\lceil m/2 \rceil$ children.
 - Every node except the root of a B tree contains at least $\lceil m/2 - 1 \rceil$ 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 $\lceil m/2 \rceil$ children whereas the leaf has 0 children.
 - All leaf nodes must be at the same level.
5. B Tree
6. At least $(n - 1)/2$ keys
7. 255
8. 11

9. 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.

10. 3

11. False

12. True

13. True

4.11 POSSIBLE QUESTIONS

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 FURTHER READINGS

Data Structures Using C by Reema Theraja Publisher: Oxford Publication