New Edition

DATA STRUCTURE AND ALGORITHMS

Arjun Singh Saud
Bhupendra Saud

Data Structure and Algorithms

B.Sc. CSIT/BIM

Arjun Singh Saud

Lecturer
Central Department of Computer Science & Information Technology
TU, Kirtipur

Bhupendra Singh Saud

IT Officer
HIDCL
Faculty Member
New Summit College
Kathmandu

KEC Publication and Distribution Pvt. Ltd.
Kathmandu, Nepal
Phone: 01-4168301, 01-4241777

DATA STRUCTURE AND ALGORITHMS

Title
DATA STRUCTURE AND ALGORITHMS
B.Sc. CSIT/BIM

ISBN No.
978 – 9937 – 724 – 09 – 8

Publishers
KEC Publication and Distribution (P.) Ltd.
Kathmandu
Ph. 01 4168301

Distributors
KEC Publication and Distribution (P.) Ltd.
Kathmandu
Ph. 01 4168301, 4241777

Copyright
Authors

Edition
New 2075

E-mail
kecpublication14@gmail.com

Price
Rs. 595.00

Preface

Data Structures has become the fundamental part in programming. We have prepared this book to fulfill the needs of the students covering most of the aspects of data structures and the logics behind these structures. It is designed to provide an introduction to data structures and algorithms, including their design, analysis, and implementation.

This book is designed as a primary text for data structure and algorithms using C as well as JAVA programming at the undergraduate level especially for **B.Sc. CSIT (TU)** and **BIM TU**. It is also convenient for BIM, BBIS, BIT, BE Computer, BCA, Bed Computer and other IT programs. It will also appeal to readers who are enrolled in a training class or who are learning advanced programming courses. This book has covered numerous practical exercises in C and JAVA. We have chosen simple problems as far as possible to make the concepts clear when they're first introduced.

This book has eleven chapters. Each chapter ends with exercises– a series of questions related to material covered in the chapter. Chapter 1 describes the overview of the C Programming. Chapter 2 gives the concept of analyzing algorithms using asymptotic notations. It also presents the overview of the data structures which are the building blocks of the computer programs. The linked list concepts are made clear in chapter 3. The linear data structures: Stack and its applications are discussed in Chapters 4. Another important data structure: Queue with its applications and variations are described in Chapter 6. Chapter 7 presents about the Tree which has become the common data structure in computer science for maintaining the file systems. Chapter 8 describes the multi way tree. Chapter 9 deals with different searching and sorting techniques and analysis of these techniques. Chapter 10 opens the picture how graphs are applied in structuring the different problems. And finally Hashing, one of the most efficient techniques in searching is discussed in chapter 11.

Authors

Acknowledgments

First, we'd like to thank all of our colleagues who inspired us to write this book. We owe a huge debt to our family members who supported us at the preparation of this book providing all the authors space and time at home.

Special thank goes to of KEC Publication and Distribution Pvt. Ltd. Family for publishing and bringing this book to market.

Students and colleagues from different colleges where we are teaching this subject also provided us valuable remarks and suggestions.

Finally, we wish to get more feedbacks and suggestions from the students and colleagues from different colleges in the coming years and help to make this book better when it comes in newer versions.

Authors
Arjun Singh Saud
Bhupendra Singh Saud

Data Structures and Algorithms

Course Title: Data Structures and Algorithms — Full Marks: 60 + 20 + 20

Course No: CSC206 — Pass Marks: 24 + 8 + 8

Nature of the Course: Theory + Lab — Credit Hrs: 3

Semester: III (B.Sc. CSIT)

Course Description: This course includes the basic foundations in of data structures and algorithms. This course covers concepts of various data structures like stack, queue, list, tree and graph. Additionally, the course includes idea of sorting and searching.

Course Objectives:

- To introduce data abstraction and data representation in memory
- To describe, design and use of elementary data structures such as stack, queue, linked list, tree and graph
- To discuss decomposition of complex programming problems into manageable sub-problems
- To introduce algorithms and their complexity

Course Contents:

Unit 1: Introduction to Data Structures & Algorithms **(4 Hrs.)**

1.1 Data types, Data structure and Abstract date type

1.2 Dynamic memory allocation in C

1.3 Introduction to Algorithms

1.4 Asymptotic notations and common functions

Unit 2: Stack **(4 Hrs.)**

2.1 Basic Concept of Stack, Stack as an ADT, Stack Operations, Stack Applications

2.2 Conversion from infix to postfix/prefix expression, Evaluation of postfix/ prefix expressions

Unit 3: Queue **(4 Hrs.)**

3.1 Basic Concept of Queue, Queue as an ADT, Primitive Operations in Queue

3.2 Linear Queue, Circular Queue, Priority Queue, Queue Applications

Unit 4: Recursion **(3 Hrs.)**

4.1 Principle of Recursion, Comparison between Recursion and Iteration, Tail Recursion

4.2 Factorial, Fibonacci Sequence, GCD, Tower of Hanoi (TOH)

4.3 Applications and Efficiency of Recursion

Unit 5: Lists **(8 Hrs.)**

5.1 Basic Concept, List and ADT, Array Implementation of Lists, Linked List

5.2 Types of Linked List: Singly Linked List, Doubly Linked List, Circular Linked List.

5.3 Basic operations in Linked List: Node Creation, Node Insertion and Deletion from Beginning, End and Specified Position

5.4 Stack and Queue as Linked List

Unit 6: Sorting **(8 Hrs.)**

6.1 Introduction and Types of sorting: Internal and External sort

6.2 Comparison Sorting Algorithms: Bubble, Selection and Insertion Sort, Shell Sort

6.3 Divide and Conquer Sorting: Merge, Quick and Heap Sort

6.4 Efficiency of Sorting Algorithms

Unit 7: Searching and Hashing **(6 Hrs.)**

7.1 Introduction to Searching, Search Algorithms: Sequential Search, Binary Search

7.2 Efficiency of Search Algorithms

7.3 Hashing: Hash Function and Hash Tables, Collision Resolution Techniques

Unit 8: Trees and Graphs **(8 Hrs.)**

8.1 Concept and Definitions, Basic Operations in Binary Tree, Tree Height, Level and Depth

8.2 Binary Search Tree, Insertion, Deletion, Traversals, Search in BST

8.3 AVL tree and Balancing algorithm, Applications of Trees

8.4 Definition and Representation of Graphs, Graph Traversal, Minimum Spanning Trees: Kruskal and Prims Algorithm

8.5 Shortest Path Algorithms: Dijksrtra Algorithm

Laboratory Works:

The laboratory work consists of implementing the algorithms and data structures studied in the course. Student should implement at least following concepts;

- Dynamic memory allocation and de-allocation strategies
- Stack operations and Queue operations
- Array and Linked List implementation of List
- Linked List implementation of Stack and Queues
- Sorting, Searching and Hashing algorithms
- Binary Search Trees and AVL Tress
- Graph Representation, Spanning Tree and Shortest Path Algorithms

Text Books:

1. Y Langsam, MJ Augenstein and A.M, Tanenbaum Data Structures using C and C++, Prentice Hall India, Second Edition 2015

Reference Books:

1. Leen Ammeral, Programmes and Data Structures in C, Wiley Professional Computting
2. G.W Rowe, Introduction to Data Structure and Algorithms with C and C++, prentice Hall India
3. R.L Kruse, B.P. Leung, C.L. Tondo, Data Structure and Program Design in C Prentice-Hall India

IT 218: Data Structure and Algorithm with Java
BIM

Course Objectives

This course aims to provide a systematic introduction to data structures and algorithms for constructing efficient computer programs. The course emphasizes on data abstraction issues (through ADTs) in the program development process, and on efficient implementation of chosen data structures and algorithms.

Laboratory work is essential in this course.

Course Description

The course contains Complexity Analysis, Linked Lists, Stacks and Queues, Recursion, Binary Trees, Multi-way Trees, Graph, Sorting, Hashing.

Course Details

Unit 1: Complexity Analysis LH 4

Computational and Asymptotic Complexity. Big-O Notation. Properties of Big-O Notation Ω and Q. Possible Problems. Examples of Complexities. Finding Asymptotic Complexity: Examples. The Best, Average, and Worst Cases 66. Amortized Complexity 69. NP-Completeness 73.

Unit 2: Linked Lists LH 5

Singly Linked Lists: Insertion, Deletion, Search. Doubly Linked Lists: Circular Lists, Skip Lists.

Self-Organizing Lists, Sparse Tables. Case Study: A Library.

Unit 3: Stacks and Queues LH 4

Stacks, Queues, Priority Queues. Case Study: Existing a Maze.

Unit 4: Recursion LH 4

Recursive Definitions. Method Calls and Recursion Implementation. Anatomy of a Recursive Call. Tail Recursion. Nontail Recursion. Indirect Recursion. Nested Recursion. Excessive Recursion. Backtracking.

Unit 5: Binary Trees LH 9

Trees, Binary Trees, and Binary Search Trees. Implementing Binary Trees. Searching a Binary Search Tree. Tree Traversal. Breadth-First Traversal. Depth-First Traversal. Insertion, Deletion, Deletion by Merging, Deletion by Copying. Balancing a Tree. The DSW Algorithm. AVL Trees. Self-Adjusting Trees. Self-Restructuring Trees, Splaying. Heaps: Heaps as Priority Queues, Organizing Arrays as Heaps, Polish Notation and Expression Trees. Operations on Expression Trees. Case Study: Computing Word Frequencies 280.

Unit 6: Multiway Trees LH 5

The Family of B-Trees. B-Trees, B*-Trees, B+-Trees. Case Study: Spell Checker

Unit 7: Graphs LH 6

Graph Representation. Graph Traversals, Shortest Paths, All-to-All Shortest Path Problem, Cycle Detection. Spanning Trees. Connectivity. Connectivity in Undirected Graphs, Connectivity in Directed Graphs. Topological Sort, Networks.

Unit 8: Sorting LH 6

Elementary Sorting AlgorithmsInsertion Sort, Selection Sort, Bubble Sort. Efficient Sorting Algorithms: Heap Sort, Quicksort, Mergesort, Radix Sort. Case Study: Adding Polynomials.

Unit 9: Hashing LH 5

Hash Functions: Division, Folding, Mid-Square Function, Extraction. Collision Resolution: Open Addressing, Chaining, Bucket Addressing, Deletion. Case Study: Hashing with Buckets.

Textbooks:

Drozdek Adam, Data Structures and Algorithms in Java, 3edition

Reference:

- Duncan A. Buell, Data Structures Using Java
- Main Michael, Data Structures and Other Objects Using Java, Prentice Hall (4 edition),
- Robert Lafore, Data Structures and Algorithms in Java, Sams Publishing;
- Narasimha Karumanchi, Data Structures And Algorithms Made Easy In Java, Career Monk Publications

Table of Contents

Chapter 1 Overview of C ---- 1

Array ---- 1
Types of Array ---- 4
Deleting of an existing element from given array ---- 15
Structure ---- 20
Unions ---- 21
Differences between structure and unions ---- 22
Pointers ---- 23
Address operator ---- 24
Dereference operator (*) ---- 24
NULL Pointers in C ---- 25
Output: ---- 26
Incrementing a Pointer ---- 30
Pointer to Arrays ---- 31
Array of pointer ---- 32
Pointers and Strings ---- 34
Dynamic Memory Allocation ---- 34
Dynamic Memory Allocation ---- 40
Functions ---- 41
Function definition ---- 42
Function call ---- 65
Exercise

Chapter 2 Complexity Analysis ---- 67

Algorithm Analysis ---- 67
Computational and Asymptotic Complexity ---- 68
Big Oh (O) notation ---- 68
Big Omega (Ω) notation ---- 69
Big Theta (Θ) notation ---- 70
Random Access Machine Model (RAM) ---- 73
Recurrences ---- 74
The Best, Average, and Worst Cases

Limitations of big-oh analysis ---- 76
Amortized Complexity ---- 76
NP-Class ---- 77
NP-Complete ---- 77
NP-complete problems Examples ---- 77
NP-hard ---- 78
What is data structure? ---- 78
Primitive Data Structure ---- 79
Non-primitive data structure ---- 80
Linear Data Structures ---- 80
Non Linear Data Structures ---- 80
Sequential representation ---- 81
Linked Representation ---- 81
Abstract Data type (ADT) ---- 82
Algorithm ---- 82
Exercise ---- 84

Chapter 3 Linked Lists

Linked List ---- 85
Self referential structure ---- 86
Header nodes ---- 87
Singly linked list ---- 87
Circular Linked list ---- 99
Doubly Linked List (DLL) ---- 106
Circular Doubly Linked List ---- 114
Skip Lists ---- 122
Algorithm ---- 123
Exercise ---- 123

Chapter 4 Stacks and Its Applications

What is stack? ---- 125
Representation of Stacks ---- 126
Applications of Stack ---- 126
The Stack ADT ---- 126
Implementation of Stack ---- 126
Array (static) implementation of a stack ---- 126
Application of stack: Delimiter Matching ---- 132

Linked list implementation of stack (or dynamic implementation of stack) ----133
Exercise ----147

Chapter 5 Recursion

Introduction ----149
Divide-and-conquer algorithms ----150
Types of recursion ----162
Direct recursion ----163
Indirect recursion ----163
Backtracking ----164
Exercise ----166

Chapter 6 Queue

What is a queue? ----167
The Queue as ADT ----168
Array implementation of queue ----168
Linear queue ----168
Linked list implementation of linear queue in C ----174
Algorithm ----174
Insert function ----174
Delete function ----175
Problems in linear queue ----179
Circular queue ----180
Declaration of a Circular Queue: ----181
Priority queue ----190
Types of priority queues ----191
Exercise ----195

Chapter 7 Tree

Introduction ----197
Binary Trees ----199
Properties of Binary tree ----199
Expression tree ----210
Algorithm for BST searching ----218
Heaps ----247
Analysis ----252

Building a Heap ---------- 252
Exercise ---------- 260

Chapter 8 Multi-way Trees

B-Trees ---------- 262
Operations on a B-Tree ---------- 263
Search Operation in B-Tree ---------- 263
Searching a B-tree Algorithm ---------- 263
Insertion Operation in B-Tree ---------- 264
Insertion into a B-tree algorithm ---------- 264
Case 2: Deletion from a non-leaf ---------- 270
Algorithm for deleting key element from B-Tree ---------- 271
B+ Trees ---------- 286
The B+ Tree structure ---------- 286
Properties of a B+ Tree ---------- 287
B* Trees ---------- 289
Applications of B* tree ---------- 289

Chapter 9 Sorting and Searching

Introduction ---------- 291
In-place ---------- 291
Selection Sort ---------- 295
Insertion Sort ---------- 298
Divide-and-conquer algorithms ---------- 300
Quick Sort ---------- 301
Searching ---------- 317
Sequential Search ---------- 317
Binary Search ---------- 318
Exercise ---------- 321

Chapter 10 Graphs

Introduction ---------- 323
Definition of directed and undirected graphs ---------- 324
Fig: An undirected graph ---------- 324
Fig: Directed multi-graph ---------- 325

Simple and multi-graphs ---- 325
Pseudo-graphs ---- 326
Mixed graph ---- 326
Depth First Traversal (DFS) ---- 336
Complete C program for DFS ---- 340
Shortest Paths ---- 342
Dynamic programming ---- 348
Recursion VS Dynamic Programming ---- 349
Floyd's Warshall Algorithm ---- 349
Spanning tree ---- 351
Properties of Spanning Tree ---- 352
Minimum spanning tree ---- 352
Kruskal's algorithm ---- 353
Pseudo code ---- 353
Analysis ---- 353

Chapter 11 Hashing

Hash Function ---- 370
Types of Hash Functions ---- 370
Division ---- 370
Folding ---- 371
Mid-Square Function ---- 371
Extraction ---- 371
Hash collision ---- 371
Collision Resolution ---- 371
Open Addressing ---- 372
Linear probing ---- 372
Complete C program for linear probing ---- 373
Double hashing ---- 374
Chaining ---- 375
Bucket Addressing ---- 376
Exercise ---- 378
Programmes ---- 381
Bibliography ---- 464
Model Questions ---- 465

Array

Array is set of
name and seq
computed add
of memory loc
point types or

Types of A

- On
- Tw
- Mu

One dimensio

The array whi
dimensional ar
single row. He

Declaring one-

data_type arra
Following are s
int age
float w
int ma
Following are s
int valu

Chapter

1

Overview of C

Array

Array is set of homogenous data items represented in contiguous memory locations using a common name and sequence of indices starting from 0. Array is a simplest data structure that makes use of computed address to locate its elements. An array size is fixed and therefore requires a fixed number of memory locations.The elements of the array can be of any valid type- integers, characters, floating-point types or user-defined types.

Types of Array

- One dimensional array
- Two dimensional array
- Multi-dimensional array

One dimensional array

The array which is used to represent and store data in a linear form is called as single or one dimensional array. Here elements of the array can be represented either as a single column or as a single row. Here only one subscript is used.

Declaring one-dimensional array

data_type array_name[array_size];

Following are some valid array declarations:

```
int age[15];
float weight[50];
int marks[100];
```

Following are some invalid array declarations in C:

```
int value[0];
```

```
float marks[0.5];
int number[-5];
```

Array Initialization (1-D)

We can initialize array in C either one by one or using a single statement. The general format of array initialization is as follows:

```
data_type array_name[size]={element1, element2,.............,element n};
```

Example: Following are some valid initialization of one dimensional array

```
int age[5]={22, 33, 43, 24, 55};
int weight[ ]={55, 6, 77, 5, 45, 88, 96, 11, 44, 32};
float a[ ]={2, 3.5, 7.9, -5.9, -8};
char section[4]={'A', 'B', 'C', 'D'};
char name[10]="Bhupendra";
```

Example: A program to read n numbers and to find the sum and average of those numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[100], i, n, sum=0;
        float avg;
        printf("Enter number of elements");
        scanf("%d", &n);
        printf("Enter %d numbers", n);
        for(i=0; i<n; i++)
        {
                scanf("%d", &a[i]);
                sum=sum+a[i];
        }
        avg=sum/n;
        printf("sum=%d\n Average=%f", sum, avg);
        getch( );
}
```

Some common operations performed in one-dimensional array

The operations that are commonly performed in one dimensional array are listed below:

- Creating of an array
- Inserting new element at required position
- Deletion of any element
- Modification of any element
- Traversing of an array
- Merging of arrays

Creating of an array

Creating of a one dimensional array means at first declaring one dimensional array and then insert elements to given array.

3	4	5	6	7	11	23	44	55	6
0	1	2	3	4	5	6	7	8	9

Example: Program for creation of an array and inserting data items to given array

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[100], i;
        printf("Enter any ten elements of an array");
        for(i=0; i<10; i++)
        {
                scanf("%d", &a[i]);
        }
        getch( );
}
```

Inserting new element at required position

This operation is used to insert new element at required position from 0 to n^{th} index of given array. Where 0th index is the first index of given array and n^{th} index is the (last+1) th index of given predefined array of size n. Here we need to shift every element one position right from last element to specified positions element.

3	4	5	6	7	11	23	44	55	6				
0	1	2	3	4	5	6	7	8	9	…	…	…	99

Insert new element 77 at position 4

3	4	5	6	77	7	11	23	44	55	6			
0	1	2	3	4	5	6	7	8	9	10	…	…	99

Algorithm

1. Start
2. Read position of element to be inserted say it be 'pos'
3. Read element to be inserted say it be 'el'
4. Swap all elements one position right from last index to pos-1
5. Now location for new element is free so set new element at that location as,
   a. Array[pos]=el
6. Increment size of an array by one as,
   n=n+1;
7. Stop

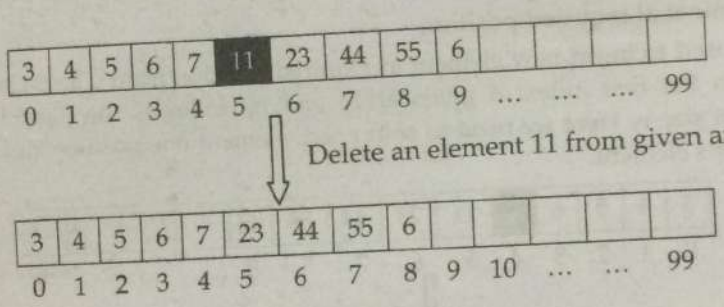
C- Function to insert new element at specified valid position of an existing array

```
void insert(int a[100], int n)
```

```
{
        int pos, nel, i;
        printf("Enter position at which you want to insert new element");
        scanf("%d", &pos);
        printf("Enter element to be inserted");
        scanf("%d", &nel);
        for(i=n-1; i>=pos; i--)
        {
                a[i+1] = a[i];
        }
        a[pos]=nel;
        n=n+1;
}
```

Deleting of an existing element from given array

This operation is used to delete any element from given array within valid position from 0th to $(n-1)^{th}$ index of given array. Where 0th index is the first index of given array and $(n-1)^{th}$ index is the index of last element of given predefined array of size n. Here we need to shift all elements one position left from index of deleted element to the index of last element and finally decrease size of given array by one.



Algorithm

1. Start
2. Read position of element to be deleted 'pos'
3. Swap all elements one position left from specified position to last element.
4. decrement size of an array by one as,

   n=n-1;
5. Stop

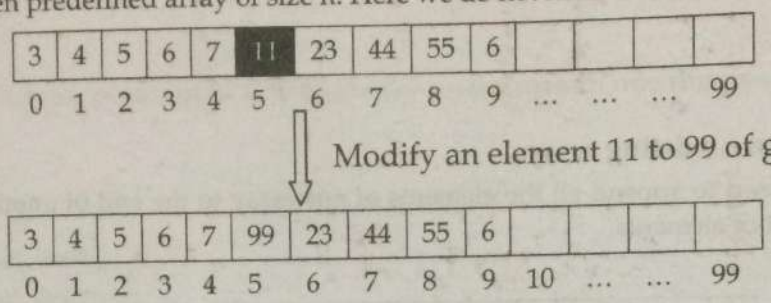
C- Function to delete element at specified valid positions from an existing array

```
void delet(int a[100], int n)
{
        int pos, i;
        printf("Enter position at which you want to delete an element");
        scanf("%d", &pos);
        for(i=pos; i<n; i++)
        {
```

```
            a[i] = a[i+1];
        }
        n=n-1;
}
```

Modification of an existing element of given array

This operation is used to modify any element from given array within valid position from 0 to $(n-1)^{th}$ index of given array. Where 0th index is the first index of given array and $(n-1)^{th}$ index is the index of last element of given predefined array of size n. Here we do not need to shift of elements.



Algorithm

1. Start
2. Read position of element to be update 'pos'
3. Read element to be updated say it be 'el'
4. Set element at given position as,
   Array[pos]=el;
5. Stop

C- Function to update element at specified valid positions of an existing array

```
void update(int a[100], int n)
{
        int pos, num;
        printf("Enter position at which you want to update an element");
        scanf("%d", &pos);
        printf("Enter new element\n");
        scanf("%d", &num);
        a[pos]=num;
}
```

Traversing of an existing array

This operation is used to display lee elements from first index to last index of given array. Here we do not need to shift of elements.

Algorithm

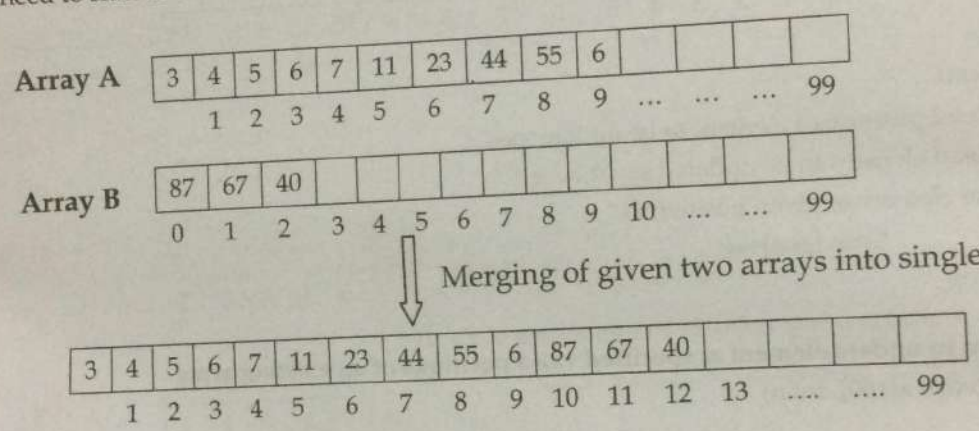
1. Start
2. Loop for i=0 to (array_size -1)
   Display Array[i]
   Increment i by one as i=i+1
3. Stop

C- Function to Traverse of an existing array

```
void traverse(int a[100], int n)
{
        int i;
        printf("Current Elements of array are:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
}
```

Merging of any two existing arrays

This operation is used to append all the elements of one array to the end of another array. Here we do not need to shift of elements.



Algorithm

1. Start
2. For i=0 to size of first array minus one
   Set all elements of first array to new array as
   New_array[i]=first_array[i];
3. Set j=size of first array
4. Loop for i=0 to size of second array minus one
   Set all elements of second array to new array as,
   New_array[j]=second_array[i]
   Increment j by one as, j=j+1
5. Increment size of new array by size of first array plus size of second array
6. Stop

C- Function to merge any two existing arrays into single common array

```
void merging(int a[100], int b[100], int n, int m)
{
  int i, j=n;
```

```
  printf("Enter element of second array");
  for(i=0; i<m; i++)
  {
    scanf("%d", &b[i]);
    a[j]=b[i];
    j++;
    n=n+1;
  }
}
```

Complete menu driven program in C to show the operations in one dimensional array

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void create(int [100], int*);
void insert(int [100], int*);
void delet(int [100], int*);
void update(int [100], int *);
void traverse(int [100], int*);
void searching(int [100], int*);
void merging(int [100], int [100], int *, int *);
int main()
{
        int a[100], b[100], m, nel, pos, i;
        int n;
        int choice;
        printf("\n Manu for program:\n");
        printf("1:Create\n2:insert\n3:delete\n4:Update\n5:Traverse\n6:searching\n7:merging\n");
        do
        {
         printf("\n Enter your choice\n");
         scanf("%d", &choice);
         switch(choice)
         {
                case 1:
                printf("Enter no of elements of first array\n");
                scanf("%d", &n);
                        create(a, &n);
                        break;
                case 2:
                        insert(a, &n);
                        break;
                 case 3:
```

```
                delet(a, &n);
                break;
        case 4:
                update(a, &n);
                break;
        case 5:
                traverse(a, &n);
                break;
        case 6:
                searching(a, &n);
                break;
        case 7:
        printf("Enter size of second array");
        scanf("%d", &m);
                merging(a, b, &n, &m);
                break;
        default:
                printf("Invalid choice");
    }
  }while(choice<8);
    return 0;
}
void create(int a[100],int *n)
{
    int i;
    printf("Enter %d elements", *n);
    for(i=0;i<*n;i++)
    {
        scanf("%d", &a[i]);
    }
}

void insert(int a[100], int *n)
{
    int pos, nel, i;
    printf("Enter position at which you want to insert new element");
    scanf("%d", &pos);
    printf("Enter new element");
    scanf("%d", &nel);
    for(i=*n-1; i>=pos; i--)
    {
        a[i+1] = a[i];
```

```
        }
        a[pos]=nel;
        *n=*n+1;
}

void delet(int a[100], int *n)
{
        int pos, i;
        printf("Enter position at which you want to delete an element");
        scanf("%d", &pos);
        for(i=pos; i<*n; i++)
        {
                a[i] = a[i+1];
        }
        *n=*n-1;
}

void update(int a[100], int *n)
{
        int pos, num;
        printf("Enter position at which you want to update an element");
        scanf("%d", &pos);
        printf("Enter new element\n");
        scanf("%d",&num);
        a[pos]=num;
}

void traverse(int a[100], int *n)
{
        int i;
        printf("Current Elements of array are:\n");
        for(i=0;i<*n;i++)
        {
                printf("%d\t", a[i]);
        }
}

void searching(int a[100], int *n)
{
        int k,i;
        printf("Enter searched item");
        scanf("%d",&k);
        for(i=0;i<*n;i++)
```

```
        {
                if(k==a[i])
                {
                        printf("Search Successful");
                        break;
                }
        }
        if(i==*n)
        printf("Search Unsuccessful");
}

void merging(int a[100], int b[100], int *n, int *m)
{
  int i, j=*n;
  printf("Enter element of second array");
  for(i=0;i<*m;i++)
  {
    scanf("%d",&b[i]);
    a[j]=b[i];
    j++;
    *n=*n+1;
  }
}
```

Two dimensional arrays

The array which is used to represent and store data in a tabular form is called as 'two dimensional array.' Such type of array specially used to represent data in a matrix form.

The following syntax is used to represent two dimensional array.

Syntax: data-type array_name [row_subscript][column-subscript];

	0	1	2
0	[0][0]	[0][1]	[0][2]
1	[1][0]	[1][1]	[1][2]

Table: Memory allocation for two dimensional array of size 2 by 3

Two dimensional array declarations

int a[3][3];

In above example, 'a' is an array of type integer which has storage size of 3 * 3 matrix. The total size would be 3 * 3 * 2 = 18 bytes if our system takes 2 byte for **int** data type.

Initialization of two dimensional array

The process of setting elements to the array during its declaration is called array initialization. The general format of array initialization in two dimensional array is:

Syntax:

data_type array_name[row_size][col_size]={element1,element2,.....,element n};

Example: Initialization of elements into two dimensional array

```
int a[2][3]={33,44,23,56,77,87};
        OR
int a[2][3]={{33,44,23},
             {56, 77, 87}};
```

Example 1: Finding transpose of given matrix

```
#include <stdio.h>
int main( )
{
        int m, n, i, j, matrix[10][10], transpose[10][10];
        printf("Enter the number of rows and columns of matrix ");
        scanf("%d%d", &m, &n);
        printf("Enter the elements of matrix \n");
        for( i = 0 ; i< m ; i++ )
        {
          for( j = 0 ; j < n ; j++ )
          {
                scanf("%d", &matrix[i][j]);
          }
        }

        for( i = 0 ; i < n ; i++ )
        {
          for( j = 0 ; j < m ; j++ )
          {
                transpose[i][j] = matrix[j][i];
          }
        }
        printf("Transpose of entered matrix :\n");
        for( i = 0 ; i < m ; i++ )
        {
          for( j = 0 ; j < n ; j++ )
          {
                printf("%d\t", transpose[i][j]);
          }
          printf("\n");
        }
        return 0;
}
```

Example 2: A program to find addition of any two matrices by using function

```
#include<stdio.h>
#include<conio.h>
void display(int [ ][ ], int, int);  //function prototype
void main( )
{
        int a[10][10], b[10][10],c[10][10],i, j, r, c;
        clrscr();
        printf("Enter size of a matrix");
        scanf("%d%d", &r, &c);
        printf("Enter elements of first matrix\n");
        for(i=0;i<r; i++)
        {
                for(j=0; j<c; j++)
                {
                        scanf("%d", &a[i][j]);
                }
        }
        printf("Enter elements of second matrix\n");
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                        scanf("%d", &b[i][j]);
                }
        }
        for(i=0;i<r;i++)  // finding sum
        {
                for(j=0; j<c; j++)
                {
                        c[i][j]=a[i][j]+b[i][j];
                }
        }
        printf("The first  matrix is\n");
        display(a, r, c);
        printf("The second  matrix is\n");
        display(b, r, c);
        printf("The resulting  matrix is\n");
        display(c, r, c);
        getch();
}
void display(int d[10][10], int r, int c)
```

```
{
        int i, j;
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                        printf("%d\t", d[i][j]);
                }
                printf("\n");
        }
}
```

Representation of elements into two dimensional array

A two dimensional array can be implemented in a programming language in two ways:

- Row-major implementation
- Column-major implementation

Row-major implementation

Row-major implementation is a linearization technique in which elements of array are reader from the keyboard row-wise i.e. the complete first row is stored, and then the complete second row is stored and so on. For example, an array a[3][3] is stored in the memory as shown in fig below:

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]
Row 1			Row 2			Row 3		

Column-major implementation

In column major implementation memory allocation is done column by column i.e. at first the elements of the complete first column is stored, and then elements of complete second column is stored, and so on. For example an array a [3] [3] is stored in the memory as shown in the fig below:

a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]	a[0][2]	a[1][2]	a[2][2]
Column 1			Column 2			Column 3		

Multi-Dimensional array

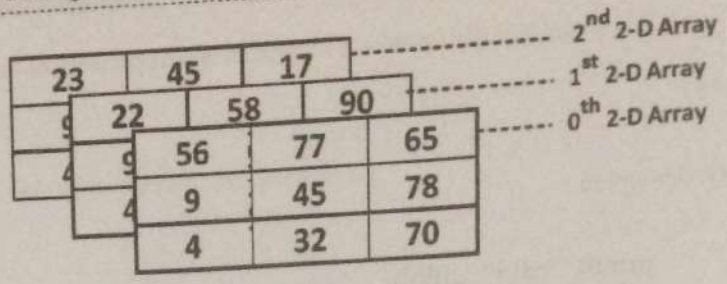
The collection of two dimensional arrays is called multi-dimensional array. C also allows arrays with more than two dimensions. For example, a three dimensional array may be declared by

```
int a[3][2][4];
```

Here, the first subscript specifies a plane number, the second subscript a row number and the third a column number.

However C does allow an arbitrary number of dimensions. For example, a six-dimensional array may be declared by

```
int b[3][4][6][8][9][2];
```



Example: Program that initialize three dimensional array and display elements

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int i, j, k;
        int arr[3][3][3]={
                {
                  {11, 12, 13},
                  {14, 15, 16},
                  {17, 18, 19}
                },
                {
                  {21, 22, 23},
                  {24, 25, 26},
                  {27, 28, 29}
                },
                {
                  {31, 32, 33},
                  {34, 35, 36},
                  {37, 38, 39}
                },
              };
                clrscr();
                printf("3D Array Elements are:\n");
                for(i=0;i<3;i++)
                {
                        for(j=0; j<3; j++)
                {
                        for(k=0; k<3; k++)
                        {
                                printf("%d\t", arr[i][j][k]);
                        }
                        printf("\n");
                }
```

```
            printf("\n");
        }
        getch();
}
```

Output

3D Array Elements:

11	12	13
14	15	16
17	18	19
21	22	23
24	25	26
27	28	29
31	32	33
34	35	36
37	38	39

Show that an array as an ADT

Let A be an array of type T and has n elements then it satisfied the following operations:

- CREATE(A): Create an array A
- INSERT(A, X): Insert an element X into an array A in any location
- DELETE(A, X): Delete an element X from an array A
- MODIFY(A, X, Y): modify element X by Y of an array A
- TRAVELS(A): Access all elements of an array A
- MERGE(A, B): Merging elements of A and B into a third array C

Structure

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name. An array is a data structure in which all the members are of the same data type. Structure is another data structure in which the individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. The individual structure elements are referred to as members. A structure is defined as,

```
struct structure_name
{
        member 1;
        member 2;
        ..........
        member n;
};
```

We can define a structure to hold the information of a student as follows:

```
struct Student
{
```

```
        char name[2];
        int roll;
        char sec;
        float marks;
};
```

Structure variable declaration

struct Student s1, s2, s3;

We can combine both template declaration and structure variable declaration in one statement.

Example: Defining structure and structure variable

```
struct Student
{
        char name[2];
        int roll;
        char sec;
        float marks;
}
s1, s2, s3;
```

Accessing members of a structure

There are two types of operators to access members of a structure. Which are:

- Member operator (dot operator or period operator (.))
- Structure pointer operator (→).

Structure initialization

Like any data type, a structure variable can be initialized as follows:

```
struct Student
{
        char name[20];
        int roll;
        char sec;
        float marks;
};
struct Student s1={"Raju", 22, 'A', 55.5};
```

The s1 is a structure variable of type Student, whose members are assigned initial values. The first member (name[20]) is assigned the string "Raju", the second member (roll) is assigned the integer value 22, the third member (sec) is assigned the character 'A', and the fourth member (marks) is assigned the float value 55.5.

Example: Program illustrates the structure in which read member elements of structure and display them.

```
#include<stdio.h>
void main()
{
```

```
struct Student
{
        char name[20];
        int roll;
        char sec;
        float marks;
};
struct Student s1;
clrscr();
printf("Enter the name of a student");
gets(s1.name);
printf("Enter the roll number of a student");
scanf("%d",&s1.roll);
printf("Enter the section of a student");
scanf("%c",&s1.sec);
printf("Enter the marks obtained by the student");
scanf("%f",&s1.marks);
//displaying the records
printf("Name=%s\n Roll number =%d\n Section=%c\n Obtained marks=%f",s1.name,
s1.roll, s1.sec, s1.marks);
}
```

Array of structures

Structure is collection of different data type. An object of structure represents a single record in memory, if we want more than one record of structure type, we have to create an array of structure or object. As we know, an array is a collection of similar type; therefore an array can be of structure type. Like any array declaration, the array of structure variable declared in similar way and is declared as follows:

```
struct Student
{
        char name[20];
        int roll;
        char sec;
        float marks;
};
struct student s[50];
```

Here, s[50] is a structure variable. It can be accommodate the structure of a student up to 50. Each record may be accessed and processed separately like individual elements of an array.

Example: Example declares an array of structure variable s having 3 members. Each member in turn is a structure having four members.

```
#include<stdio.h>
struct Student
{
```

```
        char name[20];
        int roll;
        char sec;
        float marks;
};
void main( )
{
        struct Student s[3];
        int i;
        for(i=0;i<3;i++)
        {
                printf("Enter the name of a student");
                gets(s[i].name);
                printf("Enter the roll number of a student");
                scanf("%d",&s[i].roll);
                printf("Enter the section of a student");
                scanf("%c",&s[i].sec);
                printf("Enter the marks obtained by the student");
                scanf("%f",&s[i].marks);
        }
        //displaying the records
        for(i=0;i<3;i++)
        {
                printf("Name=%s\n Roll number =%d\n Section=%c\n Obtained
                marks=%f",s[i].name, s[i].roll, s[i].sec, s[i].marks);
        }
}
```

Arrays within structures

Sometimes, arrays may be the member within structure; this is known as arrays within structure. Accessing arrays within structure is similar to accessing other members. When you find yourself to store a string value, then you have to go for array within structure because your name comes under character data type alone, thus array is capable of storing data of same data type. C allows the use of array as member of structure as below.

Example: Program in C showing array within structure

```
#include<stdio.h>
struct Student
{
        char name[20];
        int roll;
        char sec;
        float marks[5];
};
```

```
void main()
{
        struct Student s;
        int i;
        float sum=0, avg;
        printf("Enter the name of a student");
        gets(s[i].name);
        printf("Enter the roll number of a student");
        scanf("%d",&s[i].roll);
        printf("Enter the section of a student");
        scanf("%c",&s[i].sec);
        printf("Enter the marks obtained by the student in 5 subjects");
        for(i=0;i<5;i++)
        {
                scanf("%f", &s.marks[i]);
                sum=sum+s.marks[i];
        }
        avg=sum/5;
        //displaying the records
        printf("Name=%s\n Roll number =%d\n Section=%c\obtained marks=%f", s.name, s.roll,
        s.sec, avg);
}
```

Structures within Structures

Structures within structures mean nesting of structures. Study the following example and understand the concepts.

Example: Example shows a structure definition having another structure as a member. In this example, person and students are two structures. Person is used as a member of student (Person within Student).

```
#include<stdio.h>
struct Psrson
{
        char name[20];
        int age;
};
struct Student
{
        int roll;
        char sec;
        struct Person p;
};
void main()
{
```

Equivalent form of nested structure is:

```
struct Student
{
        int roll;
        char sec;
        struct Person
        {
                char name[20];
                int age;
        }p;
};
```

```
    struct Student s;
    printf("Enter the name of a student");
    gets(s.p.name);
    printf("Enter age");
    scanf("%d", &s.p.age);
    printf("Enter the roll number of a student");
    scanf("%d", &s.roll);
    printf("Enter the section of a student");
    scanf("%c", &s.sec);
    //displaying the records
    printf("Name=%s\n Roll number =%d\n Age=%d\n Section=%c\n", s.p.name, s.roll,
    s.p.age, s.sec);
}
```

Unions

A **union** is a variable that may hold objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program.

Both **structure** and **unions** are used to group a number of different variables together. Syntactically both structure and unions are exactly same. The main difference between them is in storage. In structures, each member has its own memory location but all members of union use the same memory location which is equal to the greatest member's size.

Declaration of union

The general syntax for declaring a union is:

```
union union_name
{
        data_type       member1;
        data_type       member2;
        data_type       member3;
        ..............................
        ..............................
        data_type       memberN;
};
```

We can define a **union** to hold the information of a student as follows:

```
union Student
{
        char name[2];
        int roll;
        char sec;
        float marks;
};
```

Differences between structure and unions

1. The amount of memory required to store a structure variable is the sum of sizes of all the members. On the other hand, in case of a union, the amount of memory required is the same as member that occupies largest memory.

```
#include<stdio.h>
#include<conio.h>
struct student
{
        int roll_no;
        char name[20];
};
union employee
{
        int ID;
        char name[20];
};
void main( )
{
        struct student s;
        union employee e;
        printf("\nsize of s = %d bytes", sizeof(s)); // prints 22 bytes
        printf("\nSize of e = %d bytes", sizeof(e)); // prints 20 bytes
        getch( );
}
```

The difference between structure and union are listed below:

Structure	Union
i. Access Members	
We can access all the members of structure at anytime.	Only one member of union can be accessed at anytime.
ii. Memory Allocation	
Memory is allocated for all variables	Allocates memory for variable which variable require more memory.
iii. Initialization	
All members of structure can be initialized.	Only the first member of a union can be initialized.
iv. Keyword	
'struct' keyword is used to declare structure.	'Union' keyword is used to declare union.
v. Syntax	
struct struct_name	union union_name

{ structure element 1; structure element 2; --------- --------- structure element n; } struct_var_name;	{ union element 1; union element 2; --------- --------- union element n; } union _var_name;

vi. Example

struct student
{
int roll_no;
char name[20];
};
struct student s;
	union employee
	{
	int ID;
	char name[20];
	};
	union employee e;

Pointers

In C a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer. Pointer is used in C program to access the memory and manipulate the address.

Pointers solve two common software problems. First, pointers allow different sections of code to share information easily. You can get the same effect by copying information back and forth, but pointers solve the problem better. Second, pointers enable complex "linked" data structures like linked lists and binary trees.

The unary operator & gives the 'address of a variable'. The indirection or dereference operator * gives the 'contents of an object pointed to by a pointer'.

Pointer variable declaration

A pointer is a variable that contains the memory location of another variable. Like other variables used in C the pointer variable must be declared before their use. The syntax of declaration of pointer is as shown below. The asterisk tells the compiler that you are creating a pointer variable.

Syntax: Data_type * variable name;

Example:

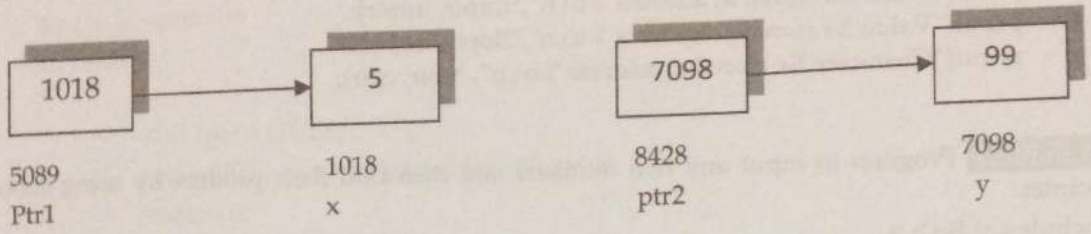
```
int *ptr1;
float *ptr2;
char *ptr3;
double *ptr4; etc
```

Initialization of pointer variable

After declaration of pointer variable it can be initialized by using assignment variable as below,

```
int x=5,y=99;
int *ptr1=&x;
int ptr2;
ptr2=&y;
```

Thus initialization of pointer variable can be done either during the declaration of pointer variable or after the declaration in next line as above.



Address operator

Once we declare a pointer variable we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

```
ptr=&num;
```

This places the address where num is stores into the variable ptr. If num is stored in memory 21260 addresses then the variable ptr has the value 21260.

Example: A program to illustrate pointer declaration

```
void main()
{
    int *ptr;
    int num;
    num=45;
    ptr=&sum;
    printf ("\n Number is %d\n", num);
    printf ("\n The Num pointer is %d", ptr);
}
```

We will get the same result by assigning the address of num to a regular (non pointer) variable. The benefit is that we can also refer to the pointer variable as *ptr the asterisk tells to the computer that we are not interested in the value 21260 but in the value stored in that memory location. While the value of pointer is 21260 the value of num is 45 however we can assign a value to the pointer * ptr as in *ptr=45.

This means place the value 45 in the memory address pointer by the variable ptr. Since the pointer contains the address 21260 the value 45 is placed in that memory location. And since this is the location of the variable num the value also becomes 45. This shows how we can change the value of pointer directly using a pointer and the indirection pointer.

Example 2: Program to display the contents of the variable their address using pointer variable

```
include< stdio.h >
void main()
{
    int num, *intptr;
    float x, *floptr;
    char ch, *cptr;
    num=123;
    x=12.34;
    ch='a';
    intptr=&x;
```

```
        cptr=&ch;
        floptr=&x;
        printf("Num %d stored at address %u\n",*intptr, intptr);
        printf("Value %f stored at address %u\n",*floptr, floptr);
        printf("Character %c stored at address %u\n", *cptr, cptr);
}
```

Example 3: Program to input any two numbers and then find their product by using concept of pointer.

```
include< stdio.h >
void main()
{
        float a, b, *p, *q, product;
        printf("Enter any two numbers");
        scanf("%f%f", &a, &b);
        p=&a;
        q=&b;
        product = *p * *q;
        printf("The product of two number is: ", product);
}
```

Difference between ordinary variable and pointer variable in C

Ordinary variable	Pointer variable
1. It stores the actual values of a variable	1. It stores the address of another variable as their value
2. Variables name directly refers to value	2. A pointer indirectly refers a value through its address
3. Ordinary variables are initialized according to their data types.	3. A pointer variable is initialized to 0, null or address of another variable.
4. It does not have void data type	4. It can have void data type
5. Its syntax is, Data_type variable_name;	5. Its syntax is, Data_type *variable_name;
6. When parameters are passed by ordinary variable, it acts as function call by value	6. When parameters are passed by pointer variable, it acts as function call by reference.

Dereference operator (*)

Using this operator we can directly access the value stored in the variable. To do this, we simply have to precede the pointer's identifier with an asterisk (*). Thus, & and * have complementary (opposite) meaning.

Eg: int *pt;

NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is

assigned NULL is called a **null** pointer. The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>
int main ()
{
        int *ptr = NULL;
        printf("The value of ptr is : %x\n", ptr );
        return 0;
}
```

Output:

The value of ptr is 0

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an 'if' statement as follows:

```
if(ptr)   /* succeeds if p is not null */
if(!ptr)  /* succeeds if p is null */
```

Pointer expressions & pointer arithmetic

There are four arithmetic operators that can be used on pointers: ++, --, +, and -.

Like other variables pointer variables can be used in expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y=*p1* *p2;
sum=sum+*p1;
z= 5 - *p2/*p1;
*p2= *p2 + 10;
p1=p2
p1=p1+1;
p2=p2-1
p1 - p2
p1==p2
```

C allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers *p1+=5; sum+=*p2; etc., we can also compare pointers by using relational operators the expressions such as p1 >p2, p1==p2 and p1!=p2 are allowed.

Example 1: Program to illustrate the pointer expression and pointer arithmetic

```
#include< stdio.h >
void main()
{
    int ptr1,ptr2;
    int a, b, x, y, z;
    a=30; b=6;
```

```
    ptr1=&a;
    ptr2=&b;
    x=*ptr1+ *ptr2 -6;
    y=6*- *ptr1/ *ptr2 +30;
    printf("\n Address of a =%u",ptr1);
    printf("\n Address of b =%u",ptr2);
    printf("\n a=%d, b=%d", a, b);
    printf("\n x=%d, y=%d", x, y);
    ptr1=ptr1 + 70;
    ptr2= ptr2;
    printf("\n a=%d, b=%d", a, b);
}
```

Example 2: Addition of integer value to pointer variable

```
#include< stdio.h >
void main()
{
    int *p, x;
    clrscr();
    x=30;
    p=&x;
    printf("Value of pointer before the operation is %u", p);
    p=p+1;   // or p++;
    printf("Value of pointer after addition operation is %u", p);
    p=p-2;  //or p-=2
    printf("Value of pointer after subtraction operation is %u", p);
    getch();
}
```

Output

The value of pointer before the operation is 8758

The value of pointer after addition operation is 8760

The value of pointer after subtraction operation is 8756

If the size of **int** data type allocated by a particular C compiler is 2 then in the above example the expression p=p+1 increase their original address by 2. Similarly p=p+2 increase their original address by 4 and so on.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

```
#include <stdio.h>
int main ()
{
 int var[] = {10, 100, 200};
 int i, *ptr;  /* let us have array address in pointer */
```

```
ptr = var;
for ( i = 0; i < 3; i++)
{
        printf("Address of var[%d] = %x\n", i, ptr );
        printf("Value of var[%d] = %d\n", i, *ptr );
        ptr++;   /* move to the next location */
}
return 0;
}
```

Output

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
#include <stdio.h>
int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;
  ptr = &var[2];
  for ( i = 3; i > 0; i--)
  {
        printf("Address of var[%d] = %x\n", i, ptr );
        printf("Value of var[%d] = %d\n", i, *ptr );
        ptr--;
  }
  return 0;
}
```

Output

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
        int  var[] = {10, 100, 200};
        int  i, *ptr;
        ptr = var;   /* let us have address of the first element in pointer */
        i = 0;
        while ( ptr <= &var[MAX - 1] )
        {
                printf("Address of var[%d] = %x\n", i, ptr );
                printf("Value of var[%d] = %d\n", i, *ptr );
                ptr++;      /* point to the next location */
                i++;
        }
        return 0;
}
```

Output

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Invalid pointer arithmetic operations

Following operations are not possible on pointer variables:

- Addition of two pointers
- Multiplication of a pointer with a constant
- Multiplication of two pointers
- Division of a pointer with a constant
- Division of two pointers

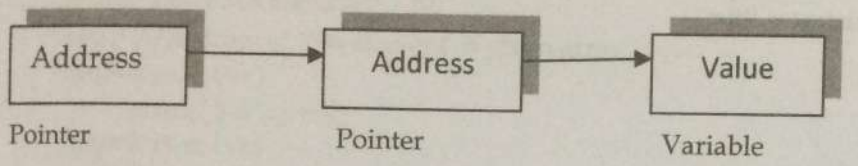
Example: Example showing invalid pointer operations

```
int *p, *q;
int x, y;
x=55, y=99;
```

p=&x;
q=&y;
p+q;
p=p*3;
p*q
p/5;
p/q etc are invalid pointer arithmetic operations.

Pointer to pointer

A pointer to a pointer is a form of multiple indirections, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type **int**:

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the **Example: Program uses pointer of pointer**

```
#include <stdio.h>
int main ()
{
        int var;
        int *ptr;
        int **pptr;
        var = 5000;
        ptr = &var;
        pptr = &ptr;
        printf("Value of var = %d\n", var );
        printf("Value available at *ptr = %d\n", *ptr );
        printf("Value available at **pptr = %d\n", **pptr);
        return 0;
}
```

Output:
Value of var = 5000
Value available at *ptr = 5000
Value available at **pptr = 5000

Pointer to Arrays

An array is actually very much like pointer. We can declare the arrays first element as a[0] or as int *a because &a[0] is an address and 'a' is also an address the form of declaration is equivalent. The difference is pointer is a variable and can appear on the left of the assignment operator that is lvalue. The array name is constant and cannot appear as the left side of assignment operator.

Consider the following:

```
int a[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to 'a', i.e. using **a[0]** through **a[5]**. But, we could alternatively access them via a pointer as follows:

```
int *ptr;
ptr = &a[0];   /* point our pointer at the first integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
#include <stdio.h>
void main()
{
        int i;
        int a[] = {1,23,17,4,-5,100};
        int *ptr;
        ptr = &a[0];  /* point our pointer to the first element of the array */
        printf("\n");
        for (i = 0; i < 6; i++)
        {
                printf("a[%d] = %d ",i, a[i]);
                printf("ptr + %d = %d\n",i, *(ptr + i));
        }
}
```

Compile and run the above program and carefully note lines A and B and that the program prints out the same values in either case.

Example 2: A program to display the contents of array using pointer

```
void main()
{
    int a[100];
    int i,j=0,n;
    printf("\nEnter number of elements of the array\n");
    scanf("%d",&n);
    printf("Enter the array elements");
    for(i=0; i< n; i++)
        scanf("%d",&a[i]);
    printf("Array element are:");
    for(ptr=a; ptr< (a+n); ptr++)
    {
        printf("Value of a[%d]=%d stored at address %u",j, *ptr, ptr);
```

```
        j++;
    }
}
```

Example 3: A program to read any n different numbers from user then find their sum and average by using the concept of pointer.

```
void main()
{
    int n, a[100];
    int i, *p;
    float sum=0.0, avg;
    printf("\nEnter size of n\n");
    scanf("%d",&n);
    printf("Enter %d elements", n);
    p=a ; //address of first element of given array
    for(i=0; i< n; i++)
        scanf("%d",(p+i));
    for(i=0; i< n; i++)
    {
        sum=sum+*(p+i);
    }
    avg=sum/n;
    printf("Sum=%f\n Average=%f", sum, avg);
}
```

Array of pointer

Like decleration of pointer variable to keep the address of another variable, we can declare an array of pointer to keep the address of all the members of given array. Following is the declaration of an array of pointers to an integer:

```
int *ptr[MAX];
```

This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
        int var[] = {10, 100, 200};
        int i, *ptr[MAX];
        for ( i = 0; i < MAX; i++)
        {
                ptr[i] = &var[i]; /* assign the address of integer. */
        }
        for ( i = 0; i < MAX; i++)
```

```
        {
                printf("Value of var[%d] = %d\n", i, *ptr[i] );
        }
        return 0;
}
```

Output

Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

We can also use an array of pointers to character to store a list of strings as follows:

```
#include <stdio.h>
const int MAX = 4;
int main ()
{
        char *names[] = {
                        "Bhupi saud",
                        "Geeta Karki",
                        "Aarav saud",
                         "Aayan saud",
                         };
        int i = 0;
        for ( i = 0; i < MAX; i++)
        {
                printf("Value of names[%d] = %s\n", i, names[i] );
        }
        return 0;
}
```

Output

Value of names[0] = Bhupi Saud
Value of names[1] = Geeta Karki
Value of names[2] = Aarav saud
Value of names[3] = Aayan saud

Pointers and Strings

The study of strings is useful to further tie in the relationship between pointers and arrays. It also makes it easy to illustrate how some of the standard C string functions can be implemented. Finally it illustrates how and when pointers can and should be passed to functions.

In C, strings are arrays of characters. This is not necessarily true in other languages. In BASIC, Pascal, FORTRAN and various other languages, a string has its own data type. But in C it does not. In C a string is an array of characters terminated with a binary zero Character (written as '\0'). To start off our discussion we will write some code which, while preferred for illustrative purposes, you would probably never write in an actual program. Consider, for example:

```
char my_string[40];
my_string[0] = 'A';
my_string[1] = 'a';
my_string[2] = 'r';
my_string[4] = 'a';
my_string[5] = 'v';
my_string[6] = '\0';
```

Since writing the above code would be very time consuming, C permits two alternate ways of achieving the same thing. First, one might write:

```
char my_string[40] = {'A', 'a', 'r', 'a', 'v', '\0',};
```

But this also takes more typing than is convenient. So, C permits:

```
char my_string[40] = "Aarav";
```

Now, consider the following program:

```
#include <stdio.h>
void main()
{
        char strA[80] = "A string to be used for demonstration purposes";
        char strB[80];
        char *pA;          /* a pointer to type character */
        char *pB;          /* another pointer to type character */
        puts(strA);        /* show string A */
        pA = strA;         /* point pA at string A */
        puts(pA);          /* show what pA is pointing to */
        pB = strB;         /* point pB at string B */
        putchar('\n'); /* move down one line on the screen */
        while(*pA != '\0')          /* line A (see text) */
        {
                *pB++ = *pA++;  /* line B (see text) */
        }
        *pB = '\0';
        puts(strB);        /* show strB on screen */
}
```

In the above we start out by defining two character arrays of 80 characters each. Then, **strA** has the first 42 characters initialized to the string in quotes.

Pointers and structures

We know the name of an array stands for the address of its 0th element the same concept applies for names of arrays of structures. Suppose item is an array variable of struct type. Consider the following declaration:

```
struct products
{
        char name[30];
```

```
    int manufac;
    float net;
}item[2],*ptr;
```

this statement declares item as array of two elements, each type struct products and ptr as a pointer data objects of type struct products, the assignment ptr=item; would assign the address of zeroth element to product[0]. Its members can be accessed by using the following notation.

```
ptr→name;
ptr→manufac;
ptr→net;
```

The symbol → is called arrow pointer and is made up of minus sign and greater than sign. Note that ptr→ is simple another way of writing product[0].

When the pointer is incremented by one it is made to pint to next record i.e. item[1]. The following statement will print the values of members of all the elements of the product array.

```
for(ptr=item; ptr< item+2;ptr++)
        printf("%s%d%f\n", ptr→name, ptr→manufac, ptr→net);
```

We could also use the notation (*ptr).number to access the member number. The parenthesis around ptr are necessary because the member operator '.' has a higher precedence than the operator *.

Dynamic Memory Allocation

Introduction

Most often we face situations in programming where the data is dynamic in nature. That is the number of data items keep changing during execution of the program. For example, consider a program for processing the list of customers of a corporation. The list grows when names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. Such situations can be handled more easily and effectively by using what is known as dynamic data structure in conjunction with dynamic memory management techniques.

Dynamic Memory Allocation

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgment of size, if it is wrong, may, cause failure of the program or wastage of memory space. Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as dynamic memory allocation. Although C does not inherently have this facility, there are four library routines known as "memory management functions" that can be used for allocating and freeing memory during program execution. They are listed in Table below. These functions help us build complex application programs that use the available memory intelligently.

Function	Task
malloc	Allocates memory requests size of bytes and returns a pointer to the Ist byte of allocated space
calloc	Allocates space for an array of elements initializes them to zero and returns a pointer to the memory
free	Frees previously allocated space
realloc	Modifies the size of previously allocated space.

Allocating a block of memory

A block mf memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr=(cast_type*)malloc(byte_size);
```

ptr is a pointer of type cast-type the malloc returns a pointer (of cast type) to an area of memory with size byte-size.

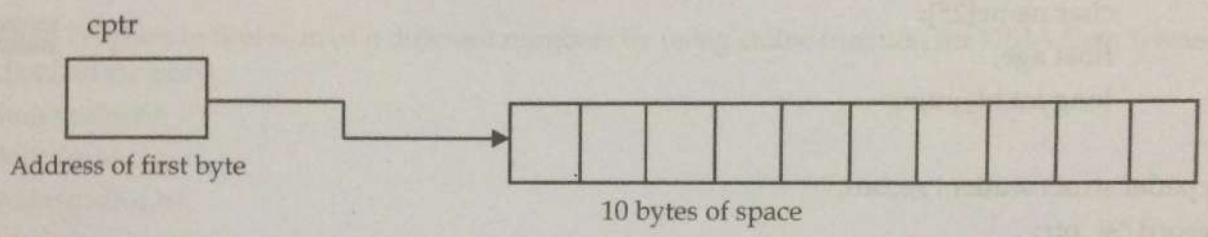
Eg.

```
x=(int*)malloc(100*sizeof(int));
```

On successful execution of this statement a memory equivalent to 100 times the area of int bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer x of type int Similarly, the statement

```
cptr = (char*)malloc(10);
```

Allocate 10 bytes of space for the pointer cptr of type char. This is illustrated below:



Example: Program to find sum of n different numbers by using **malloc** function for DMA.

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
void main()
{
        int n, i, *a, s=0;
        printf("Enter number of elements");
        scanf("%d",&n);
        a=(int*)malloc(sizeof(int)*n);
        printf("Enter %d numbers", n);
        for(i=0; i<n; i++)
        {
                scanf("%d", (a+i));
        }
        printf("Sum of given numbers is:");
        for(i=0;i<n;i++)
        {
                s=s+*(a+i);
        }
        printf("%d", s);
        getch();
}
```

Allocating multiple blocks of memory

Calloc is another memory allocation function that is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. Calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types – such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of calloc is:

```
ptr=(cast_ype*) calloc(n, elem_size);
```

The above statement allocates contiguous space for n blocks each size of elements size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space a null pointer is returned.

The following segment of a program allocates space for a structure variable:

```
struct student
{
        char name[25];
        float age;
        long int id_num;
};
typedef struct student record;
record *st_ptr;
int class_size = 30;
st_ptr = (record*) calloc (class_size, sizeof (record));
```

Example: Program to find sum of n different numbers by using **calloc** function for DMA.

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
void main()
{
        int n, i, *a, s=0;
        printf("Enter number of elements");
        scanf("%d",&n);
        a=(int*)calloc(n, sizeof(int));
        printf("Enter %d numbers", n);
        for(i=0;i<n;i++)
        {
                scanf("%d", (a+i));
        }
        printf("Sum of given numbers is:");
        for(i=0;i<n;i++)
        {
                s=s+*(a+i);
        }
```

```
        printf("%d", s);
        getch();
}
```

Releasing the used space

Compile time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic runtime allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function.

```
        free(ptr);
```

ptr is a pointer that has been created by using malloc or calloc.

Example: Program to find sum of n different numbers by using **calloc** function for DMA then release the allocated memory.

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
void main()
{
        int n, i, *a, s=0;
        printf("Enter number of elements");
        scanf("%d",&n);
        a=(int*)calloc(n, sizeof(int));
        printf("Enter %d numbers", n);
        for(i=0;i<n;i++)
        {
                scanf("%d",(a+i));
        }
        printf("Sum of given numbers is:");
        for(i=0;i<n;i++)
        {
                s=s+*(a+i);
        }
        printf("%d", s);
        free(a);
        getch();
}
```

To alter the size of allocated memory

The memory allocated by using calloc or malloc might be insufficient or excess sometimes in both the situations we can change the memory size already allocated with the help of the function realloc. This process is called reallocation of memory. The general statement of reallocation of memory is;

ptr=realloc(ptr,newsize);

This function allocates new memory space of size newsize to the pointer variable ptr ans returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region.

Example: Program for reallocation

```
#include< stdio.h >
#include< stdlib.h >
#define NULL 0
void main()
{
        char *buff;
        if((buff=(char *) malloc(10))==NULL) /*Allocating memory*/
        {
                printf("Malloc failed");
                exit(1);
        }
        strcpy(buff, "Kathmandu");
        printf("Buffer contains:%s", buff);
        if((buff=(char *)realloc(buff,15))==NULL)  /*Reallocation*/
        {
                printf("Reallocation failed");
                exit(1);
        }
        printf("Buffer size modified");
        printf("Buffer still contains: %s", buff);
        strcpy(buff, "Hastinapur");
        printf("Buffer now contains:%s\n", buff);
        free(buff);  /*freeing memory*/
}
```

Example: Program to find the product of any two matrices by using dynamic memory allocation

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int  **a, **b, **c, i, j,  m=1,r1,c1,r2,c2;
        clrscr();
        printf("Enter size of first matrix");
        scanf("%d%d",&r1, &c1);
        printf("Enter size of second matrix");
        scanf("%d%d",&r2, &cc);
        for(i=0;i<r1;i++)
        {
                *(a+i)=(int*) malloc(sizeof(int)*c1);
                *(b+i)=(int*) malloc(sizeof(int)*c1);
```

```
            *(c+i)=(int*) malloc(sizeof(int)*c1);
        }
        printf("Enter elements of first matrix:\n");
        for(i=0;i<r1;i++)
        {
            for(j=0;j<c1;j++)
            {
                scanf("%d", (*(a+i)+j));
            }
        }
        printf("Enter elements of second matrix:\n");
        for(i=0;i<r2;i++)
        {
            for(j=0;j<c2;j++)
            {
                scanf("%d", (*(b+i)+j));
            }
        }
        if(c1==r2)
        {
            for(i=0;i<r1;i++)
            {
                for(j=0;j<c2;j++)
                {
                    *(*(c+i)+j)=0;
                    for(k=0;k<c1;k++)
                    {
                        *(*(c+i)+j)= *(*(c+i)+j) + *(*(a+i)+k) * *(*(b+k)+j);
                    }
                }
            }
        }
        printf("The resultant matrix is");
        for(i=0;i<r1;i++)
        {
            for(j=0;j<c2;j++)
            {
                printf("%d\t", *(*(c+i)+j));
            }
            printf("\n");
        }
        getch();
    }
```

Functions

A function is a self- contained program segment that carries out some specific, well-defined tasks. Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. All functions are at the same level – there is no nesting. C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries.

A function in C supports the concept of modular programming techniques when our program is too complicated. The program is divided into smaller sub modules (functions) to simplify the complexity of the program.

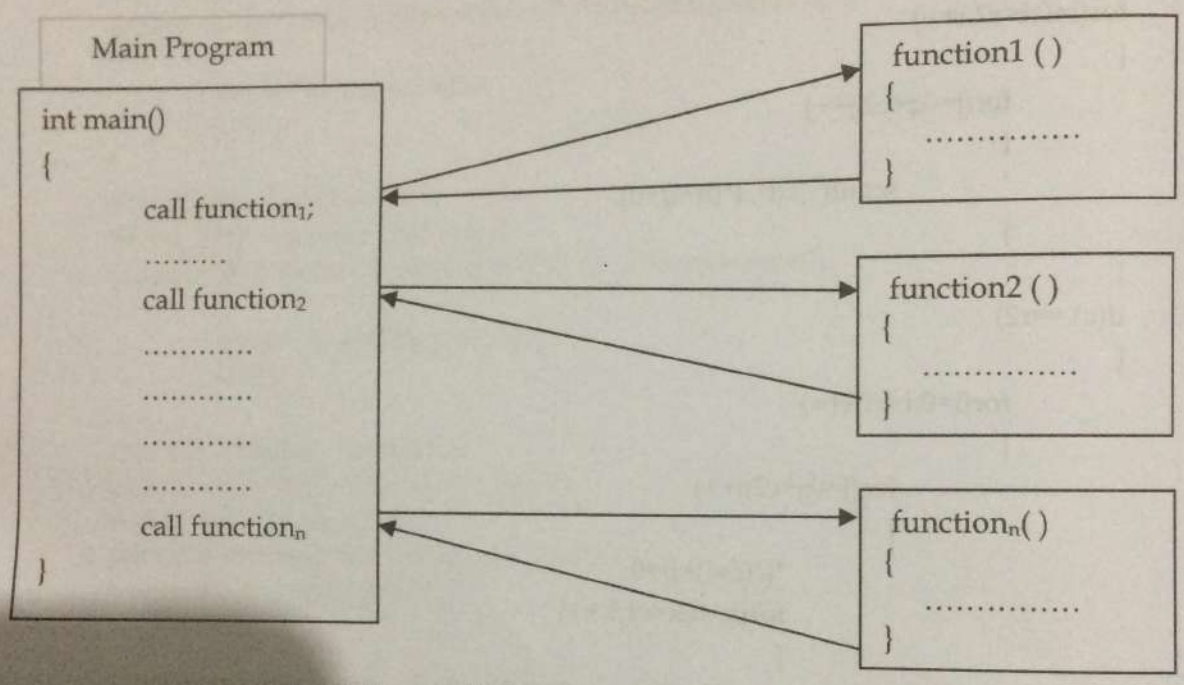


Fig: Calling multiple functions from main function

The advantages of functions are as follows

1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
2. The length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigation.
4. A function may be used by many other programs this means that a C programmer can build on what others have already done, instead of starting over from scratch.
5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.
6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program

Components of function

- Function prototype
- Function definition
- Function calling
- Function parameters
- Function return

Function declaration or prototype

A function declaration is also called a function prototype. A function may be declared at the beginning of the main function. The function declaration is of the following type:

Return_data_type function_name(formal argument$_1$, argument$_2$, – – –);

A function after execution may return a value to the function, which called it. It may not return a value at all, but may perform some operations instead. It may return an integer, character, or float. If it returns a float we may declare the function as

float function_name(float arg$_1$, int arg$_2$);

If it does not return any value we may write the above as

void fun_name(float arg1, int arg2);

If it returns a character, we may write

char fun_name(float arg1 int arg2);

If no arguments are passed into a function, an empty pair of parentheses must follow the function name.

char fun_name( );

The arguments declared as part of the prototype are also known as formal parameters. The formal arguments indicate type of data to be transferred from the calling function. This is about the function declaration.

A function declaration provides the following information to the compiler:

- The name of the function
- The type of the value to be returned (default return type is integer)
- The number of arguments that must be supplied in a function
- The type of arguments that must be supplied in a function

Function definition

A function definition has two principal components: function declarator and body of the function. The function declarator is the first line of the function definition and it must use the same function name, number of arguments, argument type and the return type same as in the function declaration but it does not have semicolon at the end. The body of the function is composed of statements enclosed within a pair of braces.

Syntax:

return_type function_name (data_type$_1$ arg$_1$, data_type$_2$ agr$_2$, ……, data_type arg$_n$)

```
{
        //body of function
}
```

A function definition in C programming language consists of a function header and a function body. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Function call

A function call is specified by the function name followed by the value of the parameters enclosed within parentheses, terminated by a semicolon.

Syntax:

Function_name (value of parameters);

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

Example: Program to show the concept of function declaration, function definition, function call, arguments, return values etc.

```
//Program to multiply any two numbers by using function
#include <stdio.h>
#include <conio.h>
int product(int, int); //Function prototype
void main( )
{
        int a, b, c;
        printf("Enter any two numbers");
        scnaf("%d%d", &a, &b);
        c=product(a, b); //function calling
        printf("The product of two number is:%d", c);
        getch( );
}
int product(int x, int y) //function definition
{
        int p;
        p=x*y;
        return(p);
```

```
}
```

Function prototype

int product(int, int);

Function prototype provides the following information to the compiler:

- The name of the function i.e. product
- The type of the value return eg int , float, void etc
- The number and the type of the arguments that must be supplied in a function call

Function definition

```
int product(int x, int y)
{
        int p;
        p=x*y;
        return(p);
}
```

The first lint of the function definition is known as function declaratory and is followed by the function body. Both decelerator and function body makeup the function definition. If the function is defined before the main function then its prototypes declaration is optional.

Function call

c=product (a, b);

A function call is specified by the function name followed by the arguments enclosed in parentheses and terminated by a semicolon. The return type is not mentioned in the function call.

Function parameters

The parameters specified in the function call are known as actual parameters and those specified in the function declaration are known as formal parameters.

E.g. In c = product (a, b); a and b are known as actual parameters and

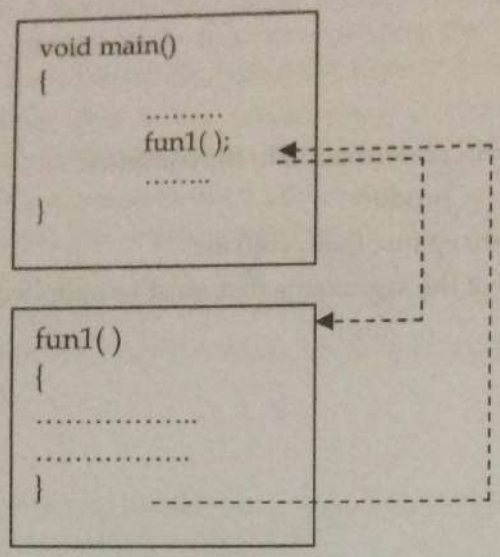
In int product (int x, int y); x and y are known as formal parameters.

Function return

The return statement tells the compiler that the execution of function is over and control should go to the calling program.

Function can be grouped into two types:

- Function that do not have a return type i.e. Void function.
- Function that do have a return value i.e. return (p).



Example: Program showing the use of non return type and return type functions

```
#include <stdio.h>
#include <conio.h>
void sum( int, int); //Function prototype of non return type function
int product(int, int); //Function prototype of return type function
void  main( )
{
        int a, b, pro;
        printf("Enter any two numbers");
        scnaf("%d%d", &a, &b);
        sum (a, b);
        pro = product (a, b); //function calling
        printf("The product of two number is:%d", pro);
        getch( );
}
void sum(int x, int y)//function definition of non return type function
{
        int s;
        s=x +y;
        printf("Sum of any two numbers=%d", s);
}
int product(int x, int y) //function definition of return type function
{
        int p;
        p=x*y;
        return (p);
}
```

Limitation of return statement in C

In C programming return statement can only return one value at a time. The statements that return multiple values such as

- return(2,5);
- return(a+2,b+5);
- Return (x, y) etc are invalid in C.

To resolve such a problem we can use the concept of call by reference discussed later.

Elimination of function declaration

If the functions are defined before they are called, the declaration of function is not necessary. In such a case first line of function definition acts as their declaration.

Example: Program to fin sum of any two number by using function without using function prototype.

```
#include<stdio.h>
#include<conio.h>
int sum(int x, int y)//function definition
{
        int s;
        s=x +y;
        return (s);
}
void main( )
{
        int a, b, s;
        printf("Enter any two numbers");
        scnaf("%d%d", &a, &b);
        s=sum (a, b);      //function calling
        printf("The sum of two number is:%d", s);
        getch( );
}
```

Example 2: A function called many times

Assume now that the program read values of 'a' & 'b' and find their sum and set to sum1, also read value of c & d and set their sum to variable sum2 and then both the sums are passed to the function to get their total. The program for doing this is given below:

```
#include<stdio.h>
void main( )
{
        float a, b, c, d, sum1, sum2, sum3;
        float add(f1oat a, float b);
        printf("enter 2 float numbers");
        scanf("%f%f", &a, &b);
        sum1 =add(a, b);
```

```
        printf("enter 2 more float numbers\n");
        scanf("%f%f", &c, &d);
        sum2 =add(c, d);
        sum3 =add(sum1, sum2);
        printf(sum of %f and %f =%f\n", a, b, sum1);
        printf("sum of %f and %f =%f\n", c, d, sum2);
        printf("sum of %f and %f =%f\n", sum1, sum2, sum3);
}

float add (float c, float d)
{
        float s;
        s = c+d;
        return s;
}
```

Categories of functions

Functions used in C programming are can be categorized into following two parts:

- Library functions and
- User defined functions

Library functions

They are built in functions and they have been already written, compiled and placed in C libraries. These functions are provided by system. printf ( ), scanf( ), getch( ), gets ( ), puts( ) etc are example of C library functions.

User defined functions

They are the functions, which are defined and used by C programmer according to their requirements.

Categories of user defined functions

Depending upon the number of arguments, types of arguments and return type of the function, the user defined functions can be categorized as below:

- Function returning value and passing arguments
- Function returning no value but passing arguments
- Function returning value and passing no arguments
- Function returning no value and passing no arguments

Function returning value and passing arguments

In this category of user defined function the value of the arguments are passed from the main function and called function return a single value to the called function as shown in following example:

```
#include <stdio.h>
#include <conio.h>
int product(int, int); //Function prototype
void main( )
```

```
{
        int a, b, c;
        printf("Enter any two numbers");
        scnaf("%d%d", &a, &b);
        c=product (a, b); //function calling
        printf("The product of two number is:%d", c);
        getch( );
}
int product(int x, int y) //function definition
{
        int p;
        p=x*y;
        return(p);
}
```

Function returning no value and passing arguments

Here the value of arguments are passed from calling function to called function but called function do not return a single value to the calling function due to which here we display the final result to the called function instate of within the main function. This category of user defined function is illustrated as shown following program:

```
#include <stdio.h>
#include <conio.h>
void product(int, int); //Function prototype
void main( )
{
        int a, b;
        printf("Enter any two numbers");
        scnaf("%d%d", &a, &b);
        product (a, b); //function calling
        getch( );
}
void product(int x, int y) //function definition
{
        int p;
        p=x*y;
      printf("The product of two number is:%d", p);
}
```

Function returning value and passing no arguments

Here values of the arguments read to the called function and from called function a single value is return to the calling function.

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

```
int product( ); //Function prototype
void main( )
{
        int c;
        c=product ( ); //function calling
        printf("The product of two number is:%d", c);
        getch( );
}
int product( ) //function definition
{
        int a, b, p;
        printf("Enter any two numbers");
        scnaf("%d%d", &a, &b);
        p=a*b;
        return (p);
}
```

Function returning no value and passing no arguments

In this category of user defined function neither value of the arguments passed from calling function to called function nor returning value from called function to calling function. Thus here reading of values of the arguments and displaying result are done within the function definition part.

```
#include <stdio.h>
#include <conio.h>
void product( ); //Function prototype
void main( )
{
        product ( ); //function calling
        getch ( );
}
void product( ) //function definition
{
        int a, b, p;
        printf("Enter any two numbers");
        scnaf("%d%d", &a, &b);
        p=a*b;
        printf("the product of two number is:%d", p);
}
```

Example: A complete C program containing more than one user defined functions

```
#include<stdio.h>
#include<conio.h>
float sum(float, float);
```

```
float difference(float, float);
float multiply(float, float);
float division(float, float);
void main( )
{
        float s, d, m, div, a, b;
        char op, flag;
        do{
        printf("Enter any operator");
        scanf("%c", &op);
        printf("Enter any two numbers");
        scanf("%f%f", &a, &b);
        switch(op)
        {
                case '+':
                        s=sum(a, b);
                        printf("Sum of two numbers=%f", s);
                        break;
                case '-':
                        d=difference(a, b);
                        printf("Difference of two numbers=%f", d);
                        break;
                case '*':
                        m=multiply(a, b);
                        printf("Product of two numbers=%f", m);
                        break;
                case '/':
                        div=division(a, b);
                        printf("Division of two numbers=%f", div);
                        break;
                default:
                        printf("Invalid operator");
        }
        printf("Enter Y for continue and N for exit");
        scanf("%c", & flag);
        }while(flag!='N');
        getch( );
}

float sum(float a, float b)
{
        return(a+b);
}
```

```
float difference(float a, float b)
{
        return(a+b);
}
float multiply(float a, float b)
{
        return(a*b);
}
float division(float a, float b)
{
        return(a/b);
}
```

Data passing mechanism to the function

Pass by value

Ordinary data types (int, float, double, char, etc) are **passed by value** in C/C++, which means that only the numerical value is passed to the function, and used to initialize the values of the functions formal parameters. Under the pass-by-value mechanism, the parameter variables within a function receive a copy of the variables (data) passed to them. Any changes made to the variables within a function are local to that function only, and do not affect the variables in main (or whatever other function called the current function) This is true whether the variables have the same name in both functions or whether the names are different. The following program illustrates 'call by value'.

Example: Program to demonstrate passing element to function by pass by value method.

```
#include<stdio.h>
#include<conio.h>
void swap(int, int ); //function prototype
void main( )
{
        int x, y;
        clrscr( );
        x = 10;
        y = 20;
        printf("Before swap values of x =%d\t y=%d", x, y);
        swap(x, y); //calling the function
        printf("After swap values of x =%d\t y=%d", x, y);
        getch( );
}
void swap(int x, int y)  //function definition
{
        int temp;
        temp = x;
        x = y;
        y = temp;
```

```
}
```

The output of the above program would be:

x = 20 y = 10

x = 20 y = 10

Pass by reference

In this method, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

```
#include<stdio.h>
#include<conio.h>
void swap(int*, int*);
void main( )
{
        int x, y;
        clrscr( );
        x = 10;
        y = 20;
        printf("Before swap values of x =%d\t y=%d", x, y);
        swap(&x, &y);
        printf("Before swap values of x =%d\t y=%d", x, y);
        getch( );
}
void swap(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

The output of the above program would be

x = 20 y = 10

x = 10 y = 20

Passing pointer to function

C programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```
#include <stdio.h>
#include <time.h>
void getSeconds(unsigned long *par);
```

```
int main ()
{
        unsigned long sec;
        getSeconds( &sec );
        printf("Number of seconds: %ld\n", sec );  /* print the actual value */
        return 0;
}

void getSeconds(unsigned long *par)
{
        *par = time( NULL );  /* get the current number of seconds */
        return;
}
```

Output

Number of seconds: 1294450468

Return pointer from functions

As we have seen in last chapter how C programming language allows to return an array from a function, similar way C allows you to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction()
{
        ........................
        ........................
}
```

Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function so you would have to define the local variable as **static** variable.

Now, consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer ie address of first array element.

```
#include <stdio.h>
#include <time.h>
int * getRandom( )        /* function to generate and retrun random numbers. */
{
        static int  r[10];
        int i;
        srand( (unsigned)time( NULL ) );
        for ( i = 0; i < 10; ++i)
        {
                r[i] = rand();
                printf("%d\n", r[i] );
        }

        return r;
```

```
}

int main ()
{
        int *p;
        int i;
        p = getRandom();
        for ( i = 0; i < 10; i++ )
        {
                printf("*(p + [%d]) : %d\n", i, *(p + i) );
        }
        return 0;
}
```

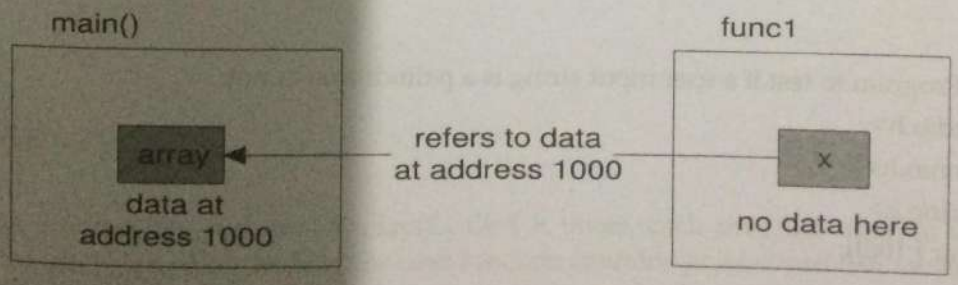
Arrays as arguments to functions

In C it is impossible to pass an entire array as an argument to a function -- instead the address of the array is passed as a parameter to the function. (In time we will regard this as a pointer). The name of an array without any index is the address of the first element of the array and hence of the whole array as it is stored contiguously. However we need to know the size of the array in the function - either by passing an extra parameter or by using the **sizeof** operator.

Example:

```
void main( )
{
        int array[20] ;
        func1(array);/* passes pointer to array to func1 */
}
```

Since we are passing the address of the array the function will be able to manipulate the actual data of the array in main( ). This is call by reference as we are not making a copy of the data but are instead passing its address to the function. Thus the called function is manipulating the same data space as the calling function.



Example 1: Program to calculate the average value of an array of doubles.

```
#include <stdio.h>
#include<conio.h>
void read_array(double [100 ], int ) ;
double average(double [100 ], int) ;
void main( )
{
```

```
        double data[ 100 ] ;
        double avg ;
        int n;
        printf("Enter no of elements");
        scanf("%d", &n);
        read_array(data, n) ;
        avg = average(data, n) ;
        printf("The average value=%lf", avg);
        getch( );
}
void read_array( double array[100 ], int size )
{
        int i ;
        for ( i = 0; i<100; i++ )
        {
                printf( "\n Enter data value array[%d] : ", i );
                scanf( "%lf", &array[i]) ;
        }
}
double average( double array[100 ], int size )
{
        double total = 0.0 ;
        int i ;
       for(i=0; i<size; i++)
        {
                total += array[i] ;
                return (total /size) ;
        }
}
```

Example 2 Program to test if a user input string is a palindrome or not.

```
#include <stdio.h>
#include <conio.h>
#include<string.h>
int palin( char [ 100]);
void main( )
{
        char str[100] ;
        puts( "Enter test string" ) ;
        gets( str ) ;
        if (palin(str))
                printf( "%s is a palindrome\n", str ) ;
```

```
        else
                printf( "%s is not a palindrome\n") ;

}
int palin( char array[100] )
{
        char temp[30] ;
        strcpy( temp, array ) ;    /* make a working copy of string */
        strrev( temp ) ;           /* reverse string */
        if (strcmp( temp, array )==0)  /* compare strings, if same strcmp returns 0 */
                return 1 ;
        else
                return 0 ;
}
```

Passing the address of array element to function

The array to a function can be passed by passing their consecutive addresses. The address of consecutive elements can be passed to a function which can be collected in a pointer variable.

```
#include<stdio.h>
#include<conio.h>
void display(int*);
void main( )
{
        int a[ ]={2,4,6,8,10,12,14,16}, i;
        for(i=0; i<8; i++)
        {
                display(&a[i]);
        }
        getch( );
}

void display(int *x)
{
        printf("%d\t", *x);
}
```

In the above program the function display() called 8 times, each time the address of individual element of the array passed to the function and function contains pointer variable as their argument and display these 8 values one by one.

Instead of passing address of each elements individually we can pass only base address of the array (the address of first element of array is called base address) to a pointer variable as an argument of a function and the elements of an array can be printed by incrementing the pointer variable as shown in program below:

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

```
void display(int*);
void main( )
{
        int a[ ]={2,4,6,8,10,12,14,16}, i;
        display(a); //which is equivalent to display(&a[0]);
        getch( );
}

void display(int *x)
{
        int i;
        for(i=0; i<8; i++)
        {
                printf("%d\t", *x);
                x++;
                //or printf("%d\t", *(x+i));
        }
}
```

Passing Multidimensional Arrays to the function

Function calls with multi-dimensional arrays will be the same as with single dimension arrays as we will still only pass the address of the first element of the array. However to declare the formal parameters to the function we need to specify size of all dimensions except the first (optional) of the array so that it may be indexed properly in the function.

Example:

2D array of doubles:	double x[10][20] ;
Call func1 with x a parameter:	func1(x);
Declaration in func1:	func1(double y[ ][20])

```
{
        ............
        //body of function
}
```

Example: A complete C program to show the concept of using multidimensional array in function. (Finding sum of any two matrices by using function)

```
#include <stdio.h>
void read(int [ ][10], char, int, int);
void sum(int [ ][10], int [ ][10], int [ ][10], int, int);
void write(int [ ][10], int, int);
int main( )
{
        int a[10][10], b[10][10], s[10][10], r, c ;
        printf("Enter size of a matrix");
        scanf("%d%d", &r, &c);
```

```
        read(a, 'A', r, c);
        read(b, 'B', r, c);
        sum(a, b, s, r, c);
        write(s, r, c);
        return 0;
}

void read(int x[ ][10], char y, int r, int c)
{
        int i, j;
        printf("Enter elements of %c matrix", y);
        for(i=0; i<r; i++)
        {
                for(j=0; j<c; j++)
                {
                scanf("%d", &x[i][j]);
                }
        }
}

void sum(int a[ ][10], int b[ ][10], int s[ ][10], int r, int c)
{
        int i, j;
        for(i=0; i<r; i++)
        {
                for(j=0; j<c; j++)
                {
                s[i][j]=a[i][j]+b[i][j];
                }
        }
}

void write(int s[ ][10], int r, int c)
{
        int i, j;
        printf("The resultant matrix is:\n");
        for(i=0; i<r; i++)
        {
                for(j=0; j<c; j++)
                {
                printf("%d\t", s[i][j]);
                }
                printf("\n");
        }
```

```
}
```

Output

```
Enter size of a matrix 3 4
Enter elements of A matrix
2 3 4 5
2 1 3 6
4 5 7 8
Enter elements of B matrix
1 2 3 4
9 8 7 6
4 5 6 1
The resultant matrix is:
3    5    7    9
11   9    10   12
8    10   13   9
```

Passing two dimensional array to a function using pointer

The two dimensional arrays with pointer can also be passed to the function exactly like one dimensional array. In one dimensional array A[size], during function call through a pointer we use *A, but in two dimensional array A[row][column] we use (*A)[column]

Where A[row] is replaced by (*A) as shown in program below:

Example: A complete C program to show the concept of passing multidimensional array in function using pointer. (Program to find the transpose of given matrix)

```
#include <stdio.h>
void transpose(int (*A)[10], int, int);
int main( )
{
        int  A[10][10], r, c , i, j;
        printf("Enter size of a matrix");
        scanf("%d%d", &r, &c);
        printf("Enter elements of given matrix');
        for(i=0; i<r; i++)
        {
                scanf("%d", (*(A+i)+j));
        }
        transpose(A, r, c);
        return 0;
}

void transpose(int (*A)[10], int r, int c)
{
        int i, j;
        printf("The transpose matrix is");
```

```
    for(i=0; i<c; i++)
    {
        for(j=0; j<r; j++)
        {
        printf("%d\t", *(*(A+i)+j));
        }
        printf("\n");
    }
}
```

Structure and functions

C supports passing of structure as arguments to function. There are two methods by which the values of a structure can be transferred from on function to another which are pass by value and pass by reference.

Passing by value

Here passing a copy of structure member or entire structure to the called function. Any change to the structure members within the function is not reflected in the original structure.

- Passing structure members to the function
- Passing entire structures to function

Passing structure members to the function

Here individual structure members are passed the functions. When a structure member is passed to the function, the value of the structure member is passed.

Example: Program to pass structure member to the function

```
#include<stdio.h>
void display(int, int);
struct sample
{
    int x;
    float y;
};
void main( )
{
    struct sample s={33, 55.5};
    display(s.x, s.y);
}
void display(int a, float b)
{
    printf("%d\n%d", a, b);
}
```

Passing entire structures to functions

Passing entire structure to the function means passing structure variable to the function.

Example: Program to pass entire structure to the function

```
#include<stdio.h>
struct student
{
        char name[20];
        int age;
        int roll;
        char sec;
};
void display(struct student);
void main( )
{
        struct student s;
        int i;
        printf("Enter the name of a student");
        gets(s.name);
        printf("Enter age");
        scanf("%d", &s.age);
        printf("Enter the roll number of a student");
        scanf("%d", &s.roll);
        printf("Enter the section of a student");
        scanf("%c", &s.sec);
        display(s);  //function call
}
void display(struct student st)
{
        //displaying the records
        printf("Name=%s\n Roll number =%d\n Age=%d\n Section=%c\n", st.name, st.roll,
        st.age, st.sec);
}
```

Passing by reference

This method uses the concept of pointer to pass structure as an argument. The address location of the structure is passed to the called functions.

```
#include<stdio.h>
void display(struct student*);
struct student
{
        char name[20];
        int age;
        int roll;
        char sec;
};
void main( )
```

```
{
        struct student s;
        int i;
        printf("Enter the name of a student");
        gets(s.name);
        printf("Enter age");
        scanf("%d", &s.age);
        printf("Enter the roll number of a student");
        scanf("%d", &s.roll);
        printf("Enter the section of a student");
        scanf("%c", &s.sec);
        display(&s); //function call
}
void sisplay(struct student *st)
{
        //displaying the records
        printf("Name=%s\n Roll number =%d\n Age=%d\n Section=%c\n", st->name,  st->roll,
         st->age, st->sec);
}
```

Returning structure variable from function

The structure variable can be return from the function. To do this, the return type of the function must be a structure type. When a structure variable is return from a function then its copy is passed to another structure variable in the calling function.

Example:

```
#include<stdio.h>
struct Distance
{
        int feet;
        int inches;
};
typedef struct Distance Dist;
Dist add (Dist, Dist);
Dist sub (Dist, Dist);
void main( )
{
        Dist d1, d2, sum, diff;
        printf("Enter feet and inches of d1 distance");
        scanf("%d%d", &d1.feer, &d1.inches);
        printf("Enter feet and inches of d2 distance");
        scanf("%d%d", &d2.feet, &d2.inches);
        sum =add(d1,d2);
        printf("Total feet=%d\n Total inches=%d", sum.feet, sum.inches);
```

```
        diff=sub(d1,d2);
        printf("Difference of feet=%d\n Difference of inches=%d", diff.feet, diff.inches);
        getch( );
}
Dist add(Dist d1, Dist d2)
{
        Dist temp;
        temp.inches=d1.inches+d2.inches;
        temp.feet=d1.feet+d2.feet+temp.inches%12;
        temp.inches=temp.inches/12;
        return temp;
}
Dist sub(Dist d1, Dist d2)
{
        Dist temp;
        if(d1.inches>d2.inches)
        {
                temp.inches=d1.inches-d2.inches;
                temp.feet=d1.feet-d2.feet;
        }
        else{
                temp.inches= d1.inches+12 - d2.inches;
                temp.feet=d1.feet- 1-d2.feet;
        }
        return temp;
}
```

Passing structure to function

This is the same case as in passing entire one dimensional array to the function. Only the difference is that, here the arguments are string (array of characters).

Example: Program to find the number of upper case letters to the given string by using the concept of passing string to function.

```
#include <stdio.h>
#include <conio.h>
int upperletter( char [100]);
void main( )
{
        char str[100] ;
        int len;
        clrscr();
        puts( "Enter test string" ) ;
        gets( str ) ;
        len=upperletter(str);
```

```
        printf("Number of upper letters in given string=%d", len);
        getch();
}
int Upperletter( char str[100] )
{
        int i, upr;
        i=0;
        while(str[i]!='\0')
        {
                i++;
                if(str[i]>='A' && str[i]<='Z')
                        upr++;
        }
        return upr;
}
```

String passing to function with pointer

This is same as passing array of character to function but only difference is that here we use pointer instead of array as below:

Example

```
#include <stdio.h>
#include <conio.h>
int upperletter( char *);
void main( )
{
        char str[100] ;
        int len;
        clrscr();
        puts( "Enter test string" ) ;
        gets( str ).;
        len=upperletter(str);
        printf("Number of upper letters in given string=%d", len);
        getch();
}
int Upperletter( char *str )
{
        int i, upr;
        i=0;
        while(*(str+i)!='\0')
        {
                i++;
                if(*(str+i)>='A' && *(str+i)<='Z')
                        upr++;
```

```
        }
        return upr;
}
```

Typedef

A typedef in C is a declaration. Its purpose is to create new types from existing types; whereas a variable declaration creates new memory locations. Since a typedef is a declaration, it can be intermingled with variable declarations, although common practice would be to state typedef first, then variable declarations. A nice programming convention is to capitalize the first letter of a user-defined type to distinguish it from the built-in types, which all have lower-case names. Also, typedefs are usually global declarations.

The general form of typedef is:

Typedef data_type variable;

Example: Use a Typedef to Create a Synonym for a Type Name

```
typedef int Integer;        //Integer can now be used in place of int
int a, b, c, d;             //4 variables of type int
Integer e, f, g, h;  //the same thing
```

In general, a typedef should never be used to assign a different name to a built-in type name; it just confuses the reader. Usually, a typedef associates a type name with a more complicated type specification, such as an array. A typedef should always be used in situations where the same type definition is used more than once for the same purpose.

Example: Program show that the use of typedef

```
#include<stdio.h>
void display(int, int);
struct sample
{
        int x;
        float y;
};
void main( )
{
        typedef struct sample st;
        st  s={66, 87.9};
        display(s.x, s.y);
}
void display(int a, float b)
{
        printf("%d\n%d", a, b);
}
```

Exercise

1. What is pointer? What are advantages of using pointer over normal variable?
2. Describe the following with example?
   (a) Pointer arithmetic
   (b) Pointer and strings
3. What are the advantages of **malloc** and **calloc**?
4. Write a program in C to find sum of any two matrices by using dynamic memory allocation.
5. Write a function to swap two float variables using call by reference.
6. Write a program for binary search using pointers.
7. Explain pointers and two-dimensional arrays with examples.
8. What is function? Describe components of function.
9. What is DMA? Describe DMA process with example.
10. Write a program in C to show the simple structure of a function
11. Write a program in C to find the sum of the series 1!/1+2!/2+3!/3+4!/4+5!/5 using the function.
12. Write a program in C to print all perfect numbers in given range using the function.
13. Write a C program to read the value of an integer m and display the value of n is 1 when m is larger than 0, 0 when m is 0 and -1 when m is less than 0.
14. Write a C program to read roll no, name and marks of three subjects and calculate the total, percentage and division.
15. Write a program in C to separate the individual characters from a string.
16. Write a program in C to count the total number of words in a string.
17. Write a program in C to find maximum occurring character in a string.
18. Write a program in C to make such a pattern like right angle triangle with a number which will repeat a number in a row.

    1
    22
    333
    4444
19. Write a program in C to find the sum of the series [1-X^2/2!+X^4/4!- .....].
20. Write a program in C to display the sum of the series [9 + 99 + 999 + 9999 ...].

•••

Chapter

2

Complexity Analysis

Algorithm Analysis

An algorithm is a clearly specified set of instructions that a computer will follow to solve the problem. Once an algorithm is given for a problem and determined to be correct, the next step is to determine the amount of resources, such as time and space that the algorithm will require. This step is called algorithm analysis. An algorithm that requires several hundred gigabytes of main memory is not useful for most current machines, even if it is completely correct.

The amount of time that any algorithm takes to run almost always depends on the amount of input that it must process. We expect, for instance, that sorting 10,000 elements requires more time than sorting 10 elements. The running time of an algorithm is thus a function of the input size. The exact value of the function depends on many factors, such as the speed of the host machine, the quality of the compiler, and in some cases, the quality of the program.

Computational and Asymptotic Complexity

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form which produces the basic nature of that algorithm. We use that general form for analysis process.

Complexity analysis of an algorithm is very hard if we try to analyze exact. We know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input. So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier. For this purpose, we need the concept of asymptotic notations. The figure below gives upper and lower bound concept.

Why asymptotic notations important?

- They give a simple characterization of an algorithm's efficiency.
- They allow the comparison of the performance of various algorithms.

Big Oh (O) notation

When we have only asymptotic upper bound then we use O notation. If f and g are any two functions from set of integers to set of integers then function f(x) is said to be big oh of g(x) i.e. f(x)=O(g(x))) if and only if there exists two positive constants c and x_0 such that for all x >= x_0, f(x) <= c*g(x)

The above relation says that g(x) is an upper bound of f(x)

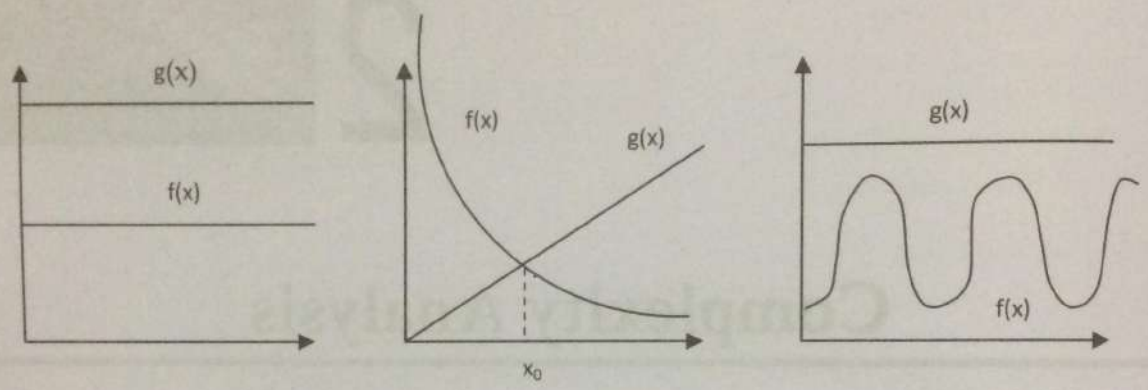


Fig: Geometric intrepretation of Big-Oh notation

Some properties

Transitivity: f(x) = O(g(x)) & g(x) = O(h(x)) then f(x) = O(h(x))

Reflexivity: f(x) = O (f(x))

Transpose symmetry: f(x) = O(g(x)) if and only if g(x)= Ω(f(x))

O (1) is used to denote constants.

For all values of n >= n0, plot shows clearly that f(n) lies below or on the curve of c*g(n)

Example: Find big oh of given function $f(n) = 3n^2 + 4n + 7$

Solution: we have $f(n) = 3n^2 + 4n + 7 <= 3n^2 + 4n^2 + 7n^2 <= 14n^2$

$f(n) <= 14n^2$

where, c=14 and $g(n) = n^2$, thus $f(n) = O(g(n)) = O(n^2)$

Big Omega (Ω) notation

Big omega notation gives asymptotic lower bound. If f and g are any two functions from set of integers to set of integers, then function f(x) is said to be big omega of g(x) i.e. f(x)=Ω(g(x)) if and only if there exists two positive constants c and x_0 such that

For all x >= x_0, f(x)>=c*g(x)

The above relation says that g(x) is a lower bound of f(x).

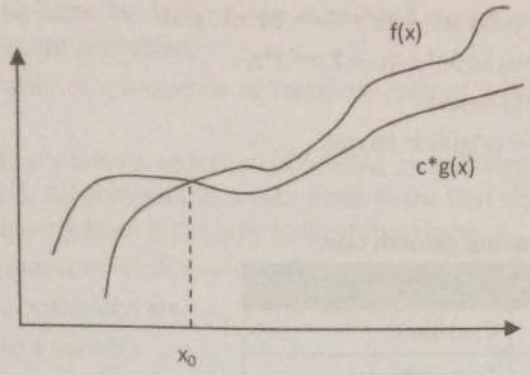


Fig: Geometric intrepretation of Big Omega notation

Some properties

Transitivity: $f(x) = \Omega(g(x))$ & $g(x) = \Omega(h(x))$ then $f(x) = \Omega(h(x))$

Reflexivity: $f(x) = \Omega(f(x))$

Example: Find big omega of $f(n) = 3n^2 + 4n + 7$

Solution: Since we have $f(n) = 3n^2 + 4n + 7 >= 3n^2$

⇨ $f(n) >= 3n^2$

where, c=3 and $g(n) = n^2$, thus $f(n) = \Omega(g(n)) = \Omega(n^2)$

Big Theta (Θ) notation

When we need asymptotically tight bound then we use this notation. If f and g are any two functions from set of integers to set of integers then function f(x) is said to be big theta of g(x) i.e. $f(x) = \Theta(g(x))$ if and only if there exists three positive constants c1, c2 and x_0 such that for all $x >= x_0$, $c1*g(x) <= f(x) <= c2*g(x)$

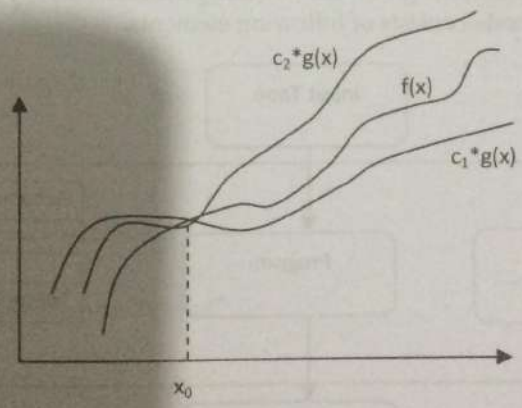


Fig: Geometric intrepretation of Big Theta notation

Some properties

Transitivity: $f(x) = \Theta(g(x))$ & $g(x) = \Theta(h(x))$ then $f(x) = \Theta(h(x))$

Reflexivity: $f(x) = \Theta(f(x))$

Symmetry: $f(x) = \Theta(g(x))$ if and only if $g(x) = \Theta(f(x))$

Example: If $f(n) = 3n^2 + 4n + 7$ $g(n) = n^2$, then prove that $f(n) = \Theta(g(n))$.

Proof: let us choose c_1, c_2 and n_0 values as 14, 1 and 1 respectively then we can have,

$f(n) <= c_1*g(n), n>=n_0$ as $3n^2 + 4n + 7 <= 14*n^2$, and

$f(n) >= c_2*g(n), n>=n_0$ as $3n^2 + 4n + 7 >= 1*n^2$

For all $n >= 1$(in both cases).

So $c_2*g(n) <= f(n) <= c_1*g(n)$ is trivial.

Hence $f(n) = \Theta(g(n))$.

Functions in order of increasing growth rate

Function	Name
C	Constant
log n	Logarithmic
$\log_2 n$	Log-squared
n	Linear
n log n	n log n
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Random Access Machine Model (RAM)

This RAM model is the base model for our study of design and analysis of algorithms to have design and analysis in machine independent scenario. In this model each basic operation (+, -) takes one step, loops and subroutines are not basic operations. Each memory reference takes one step. We measure run time of algorithm by counting the steps.

In computing time complexity, one good approach is to count primitive operations. This approach of simply counting primitive operations gives rise to a computational model called the Random Access Machine (RAM). The RAM mode consists of following elements.

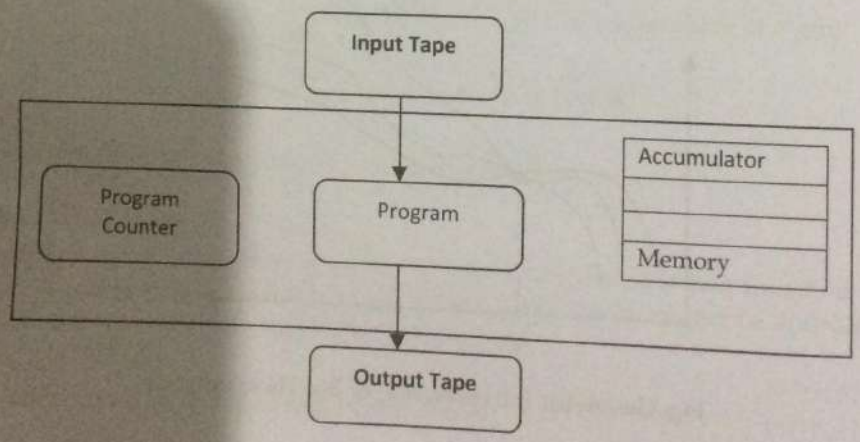


Fig: RAM Model

Input tape/output tape: input tape consists of a sequence of squares, each of which can store integer. Whenever one square is read from the tape head moves one square to the right. The output tape is also a sequence of squares, in each square an integer can be written. Output is written on the square

under the tape head and the after the writing, the tape head moves one square to the right over writing on the same square is not permitted.

Memory: The memory consists of a sequence of registers, each of which is capable of holding an integer.

Program: Program for RAM contains a sequence of labeled instructions resembling those found in assembly language programs. All computations take place in the first register called accumulator. A RAM program defines a mapping from input tape to the output tape.

Program counter: The program counter determines the next instruction to be executed.

Some examples of primitive operations are:

- Assigning a value to a variable
- Performing an arithmetic operation
- Indexing into an array
- Returning from method
- Calling a method
- Comparing two numbers etc

Detailed analysis

We can analyze iterative algorithms by counting steps by using RAM model as below.

Example 1: Algorithm for sequential search

```
Sequential_Search(A, n, key)
{
        int i, flag=0;
        for(i=0;i<n;i++)
        {
                if(A[i]==key)
                {
                        flag=1;
                }
        }
        if(flag==1)
                Print"Search successful"
        else
                Print"Search un-successful"
}
```

Space complexity

By counting number of memory references gives the space complexity of given algorithm.

The variable 'i' takes 1 unit space

The variable 'n' takes 1 unit space

The variable 'key' takes 1 unit space

The variable 'flag' takes 1 unit space

The variable 'A' takes n unit space

Now total space complexity is;

$T(n)=1+1+1+1+n$

$=4+n$

$=O(n)$

Time complexity

By counting number of statements takes the total time complexity of given algorithm.

Declaration statement takes 1 step time

In for loop;

i=0 takes 1 step time

i<n takes (n+1) step time

i++ takes n step time

Within for loop if condition takes n step time

Within if statement flag=0 takes at most n step time

If statement outside the for loop takes 1 step time

Print statement takes 1 step time

Now total time complexity is given by;

$T(n)=1+1+(n+1)+n+n+n+1+1$

$=5+4n \leq 5n+4n$

$=9n$

⇨ $T(n)=O(n)$

Example 2: Detailed analysis of Bubble sort

```
Bubble_Sort(A, n)
{
        for (i=1; i<=n; i++)
        {
                for (j=0; j<n-i; j++)
                {
                        if(A[j]>A[j+1])
                        {
                                temp=A[j];
                                A[j]=A[j+1];
                                A[j+1] = temp;
                        }
                }
        }
}
```

Analysis

Space complexity

Counting number of memory locations gives space complexity of given algorithm.

Space complexity=1+1+1+1+n=n+4=O (n) +O (1) = O (n)

Time complexity

Counting number of statements gives the total time complexity of given algorithm.

Within first for loop:

i=1 takes 1 step---------→ 1

j<=n takes (n+1) steps---------→ (n+1)

i++ takes n steps---------→ n

Within second for loop:

j=0 takes n step---------→ 1

j<n-i takes [n + (n-1) + (n-2) + --------- + 2 + 1]

j++ takes [(n-1) + (n-2) + --------- + 2 + 1]

In if statement:

It takes at most 3* (n-1)

So total time complexity (T. C) =1+(n+1) + n+[n+[n+(n-1) + (n-2) +.....+3+2+1] +

[(n-1) + (n-2) +.....+3+2+1]] + 3*[(n- 1) + (n-2) +.....+3+2+1]

$= 2n+2+ [n + n (n+1)/2+n (n-1)/2] + 3*n (n-1)/2$

$= 2n+2+n+n^2/2+n/2+ n^2/2-n/2+2n^2/2-3n/2$

$= (5n^2 + 5n)/2$

$= (O (1) * O(n^2) + O (1) * O(n))/O (1)$

$= O (n^2)$

Recurrences

To analyze the Recursive algorithms, we must need to find their recurrence relations. A recurrence relation is an equation or inequality that describes a problem in terms of itself. There are various methods are used to solve the recurrence relations which are:

- Iteration method
- Recursion tree method
- Substitution method
- Master method etc.

Example: Recursive algorithm for finding factorial

T(n)=1 when n =1

T(n)=T(n-1) + O(1) when n>1

Recursive algorithm for finding n^{th} Fibonacci number

T(1)=1 when n=1

T(2)=1 when n=2

T(n)=T(n-1) + T(n-2) +O(1) when n>2

Recursive algorithm for binary search

T(1)=1 when n=1

T(n)=T(n/2) + O(1) when n>1

Solving recurrence relation by using Iteration method

Here we expand the given relation until the boundary is not meet. Expand the relation so that summation independent on n is obtained.

Example 1: solve following recurrence relation by using iteration method

$T(n)= 2T(n/2) +1$ when $n>1$

$T(n)= 1$ when $n=1$

Solution: $T(n) = 2T(n/2) +1$

$= 2\{2T(n/4) + 1\} +1$

$= 2^2T(n/2^2) + 2 + 1$

$= 2^2\{2T(n/2^3) +1\} + 2 + 1$

$= 2^3T(n/2^3) + 2^2 + 2 + 1$

.....

.....

$= 2^k T(n/2^k) + 2^{k-1} + \ldots\ldots\ldots\ldots + 4 + 2 + 1.$

For simplicity assume:

$n/2^k = 1$

or, $n=2^k$

Taking log on both sides,

$\log n = \log 2^k$

$\log n = k \log 2$

$k = \log n$ [since $\log 2=1$]

Now, $T(n)= 2^k T(n/2^k) + 2^{k-1} + \ldots\ldots\ldots\ldots + 4 + 2 + 1.$

$T(n)= 2^k T(1) + 2^{k-1} + \ldots\ldots\ldots\ldots + 2^2 + 2^1 + 2^0$

$T(n)= (2^{k+1} -1)/ (2-1)$

$T(n)= 2^{k+1} -1 = 2.2^k -1$

$T(n)= 2n-1$

$T(n)= O(n)$

In brief, meanings of the various growth functions

Mathematical Expression	Relative Rates of Growth	Name
$T(n) = O(F(n))$	Growth of $T(n)$ is $\leq$ growth of $F(n)$	Big oh
$T(n) = \Omega(F(n))$	Growth of $T(n)$ is $\geq$ growth of $F(n)$	Big omega
$T(n) = \Theta(F(n))$	Growth of $T(n)$ is = growth of $F(n)$	Big theta
$T(n) = o(F(n))$	Growth of $T(n)$ is < growth of $F(n)$	Little oh

The Best, Average, and Worst Cases

The least possible execution time taken by an algorithm for a particular input is known as best case. **Best case complexity** gives lower bound on the running time of the algorithm for any instance of input. This indicates that the algorithm can never have lower running time than best case for particular class of problems.

Worst case complexity: The maximum possible execution time taken by an algorithm for a particular input is known as worst case. It gives upper bound on the running time of the algorithm for all the instances of the input. This insures that no input can overcome the running time limit posed by worst case complexity.

Average case complexity: It gives average number of steps required on any instance of the input.

Example: **Let's take an algorithm for Quick sort**

```
QuickSort(A,l,r)
{
        if(l<r)
        {
                p = Partition(A, l, r);
                QuickSort(A, l, p-1);
                QuickSort(A,p+1,r);
        }
}
Partition(A, l, r)
{
        x =l;
        y =r;
        p = A[l];
        while(x<y)
        {
                while(A[x] <= p)
                {
                        x++;
                }
                while(A[y] >=p)
                {
                        y--;
                }
                if(x<y)
                        swap(A[x], A[y]);
        }
        A[l] = A[y];
        A[y] = p;
        Return y;       //return position of pivot
}
```

Best Case Time Complexity

It gives best case when whole problem is divided into two partitions of equal size, therefore, $T(n) = 2T(n/2) + O(n)$, Solving this recurrence we get,

⇨ Time Complexity = $O(n \log n)$

Worst Case Time Complexity

When array is already sorted or sorted in reverse order, one partition contains n-1 items and another contains zero items, therefore its recurrence relation is,

$T(n) = T(n-1) + O(1)$, Solving this recurrence we get

⇨ Time Complexity = $O(n^2)$

Average Case Time Complexity

All permutations of the input numbers are equally likely. On a random input array, we will have a mix of well balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree suppose we are alternate: Balanced, Unbalanced, Balanced.

$B(n) = 2UB(n/2) + \Theta(n)$ Balanced

$UB(n) = B(n-1) + \Theta(n)$ Unbalanced

Solving: $B(n) = 2(B(n/2-1) + \Theta(n/2)) + \Theta(n)$

$= 2B(n/2-1) + \Theta(n)$

$= \Theta(n \log n)$

Limitations of big-oh analysis

Big-Oh analysis is a very effective tool, but it does have limitations. As already mentioned, its use is not appropriate for small amounts of input. For small amounts of input, use the simplest algorithm. Also, for a particular algorithm, the constant implied by the Big-Oh may be too large to be practical. For example, if one algorithm's running time is governed by the formula $2N \log N$ and another has a running time of 1000N, then the first algorithm would most likely be better, even though its growth rate is larger. Large constants can come into play when an algorithm is excessively complex. They also come into play because our analysis disregards constants and thus cannot differentiate between things like memory access (which is cheap) and disk access (which typically is many thousand times more expensive). Our analysis assumes infinite memory, but in applications involving large data sets, lack of sufficient memory can be a severe problem.

Sometimes, even when constants and lower-order terms are considered, the analysis is shown empirically to be an overestimate. In this case, the analysis needs to be tightened.

Amortized Complexity

Amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive. It refers to finding the average running time per operation over a worst-case sequence of operations.

Amortized analysis differs from average case performance in that probability is not involved; amortized analysis guarantees the time per operation over worst case performance. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time.

There are several techniques are used in amortized analysis:

- Aggregate analysis
- Accounting method and
- Potential method

Aggregate analysis determines the upper bound T(n) on the total cost of a sequence of n operations, then calculations the average cost to be T(n)/n.

Accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations.

Potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.

Classes P, NP and NP Complete Problems

Class P Problem

The class P consists of those problems that can be solved by a deterministic Turing machine in Polynomial time. P problems are obviously tractable. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.

Example: Adding two numbers is really easy. Surely, as the number gets larger the computation becomes harder to us human. But to a computer adding large numbers are fairly simple. We can say computers can add two numbers in Polynomial time. These types of problem which can be solved in polynomial time by a computer are known as **P problems**.

NP-Class

NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time). The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct. Every problem in this class can be solved in exponential time using exhaustive search.

Example: Let's take a little complex problem like prime factorization. We know that every composite number can be expressed as a product of two or more prime factors. Our normal PCs can handle this problem within seconds for numbers up to a billion. But as the numbers grow this problem becomes lot harder even for the fastest of computers. So this is not **solvable in Polynomial time.** So factorization is not solvable in polynomial time but the solution is verifiable in polynomial time. These problems are known as NP problems.

NP-Complete

NP-Complete problem is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time. NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:

- L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
- Every problem in NP is reducible to L in polynomial time

Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Y is reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.

NP-complete problems Examples

An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphic if one can be transformed into the other simply by renaming vertices'. Consider these two problems:

- **Graph Isomorphism:** Is graph G1 isomorphic to graph G2?
- **Sub-graph Isomorphism:** Is graph G1 isomorphic to a sub-graph of graph G2?

The Sub-graph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is in NP. This is an example of a problem that is thought to be hard, but is not thought to be NP-complete.

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list below contains some well-known problems that are NP-complete when expressed as decision problems.

- Boolean Satisfiability problem (SAT)
- Knapsack problem
- Hamiltonian path problem
- Traveling salesman problem (decision version)
- Sub-graph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem

NP-hard

Intuitively, these are the problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems. The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

Example: The halting problem is an NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.

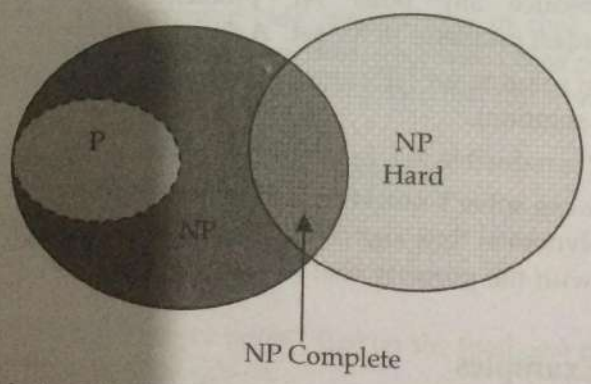


Fig: Relationships between class p, NP, NP Complete and NP Hard

What is data structure?

The data structure is basically a technique of organizing and storing of different types of data items in computer memory. It is considered as not only the storing of data elements but also the maintaining of the logical relationship existing between individual data elements. It can also be defined as a mathematical or logical model, which relates to a particular organization of different data elements.

Thus, a data structure is the portion of memory allotted for a model, in which the required data can be arranged in a proper fashion. Data structure mainly specifies the following four things:

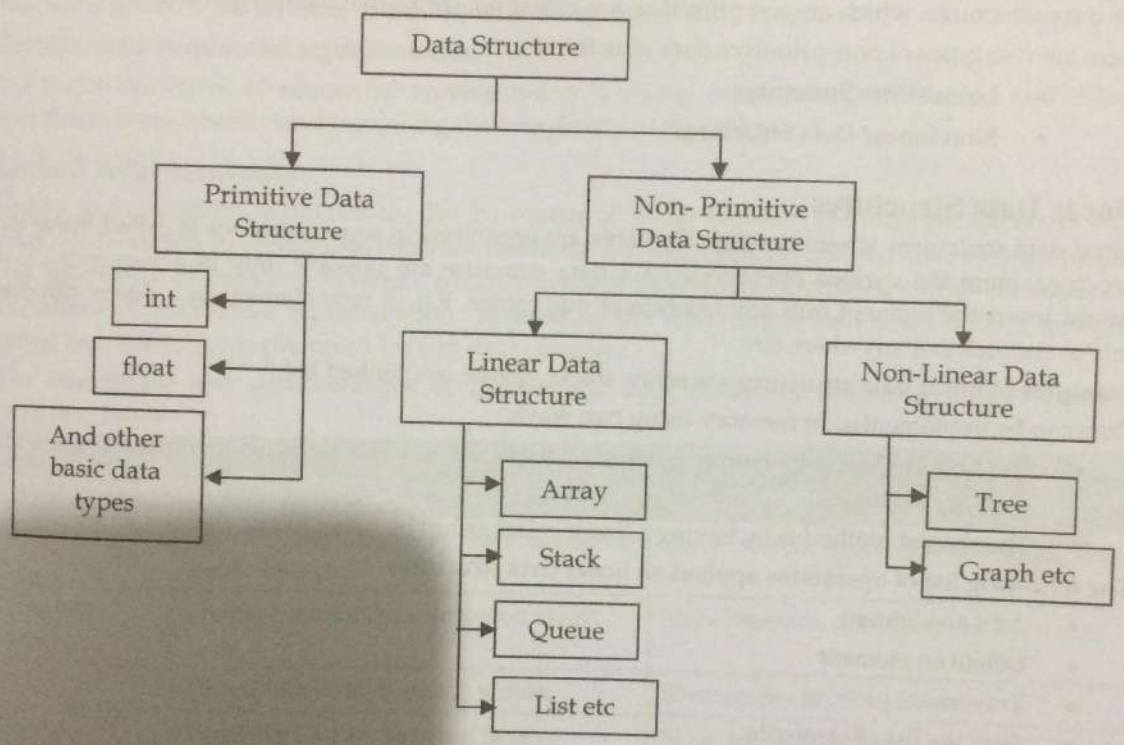
- Organization of data.
- Accessing methods
- Degree of associativity
- Processing alternatives for information

To develop a program of an algorithm, we should select an appropriate data structure for that algorithm. Therefore, algorithm and its associated data structures form a program.

Algorithm + Data structure = Program

Data Structure Classification

The classification of data structure mainly consists of:



Primitive Data Structure

The primitive data structures are known as basic data structures. These data structures are directly operated upon by the machine instructions. Normally, primitive data structures have different representation on different computers.

Example of primitive data structure:

- Integer
- Float
- Character
- Pointer

The data structures, typically those data structure that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double, etc in case of C programming are known as primitive data structures.

Non-primitive data structure

The non-primitive data structures are highly developed complex data structures. Basically, these are developed from the primitive data structure. The non-primitive data structure is responsible for organizing the group of homogeneous and heterogeneous data elements.

Example of Non-primitive data structure:

- Arrays
- Lists
- Files

The data structures, which are not primitive, are called non-primitive data structures.

There are two types of non-primitive data structures:

- Linear Data Structures
- Non Linear Data Structures

Linear Data Structures

Those data structures where the data elements are organized in some sequence is called linear data structure. Here the various operations on a data structure are possible only in a sequence i.e. we cannot insert the element into any location of our choice. E.g. A new element in a queue can come only at the end, not anywhere else.

Examples of linear data structures are array, stacks, queue, and linked list.

They can be implemented in memory using two ways:

- The first method is by having a linear relationship between elements by means of sequential memory locations.
- The second method is by having a linear relationship by using links.

The following list of operations applied on linear data structures

- Add an element
- Delete an element
- Traverse
- Sort the list of elements
- Search for a data element

Non Linear Data Structures

When the data elements are organized in some arbitrary function without any sequence, such data structures are called non-linear data structures. Examples of such type are trees, graphs. The relationship of adjacency is not maintained between elements of a non-linear data structure.

The following list of operations applied on non-linear data structures.

- Add elements
- Delete elements
- Display the elements
- Sort the list of elements
- Search for a data element

Representation of Data Structures

Any data structure can be represented in two ways. They are:

- Sequential representation
- Linked representation

Sequential representation

A sequential representation maintains the data in continuous memory locations which takes less time to retrieve the data but leads to time complexity during insertion and deletion operations. Because of sequential nature, the elements of the list must be freed, when we want to insert a new element or new data at a particular position of the list. To acquire free space in the list, one must shift the data of the list towards the right side from the position where the data has to be inserted. Thus, the time taken by CPU to shift the data will be much higher than the insertion operation and will lead to complexity in the algorithm. Similarly, while deleting an item from the list, one must shift the data items towards the left side of the list, which may waste CPU time.

Drawback of Sequential representation

The major drawback of sequential representation is taking much time for insertion and deletion operations unnecessarily and increasing the complexity of algorithm.

Linked Representation

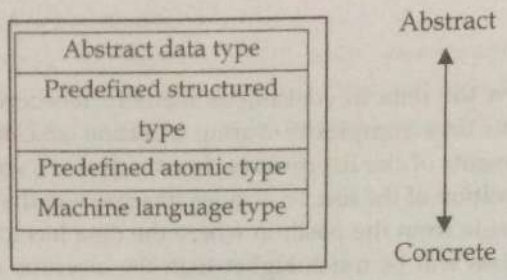
Linked representation maintains the list by means of a link between the adjacent elements which need not be stored in continuous memory locations. During insertion and deletion operations, links will be created or removed between which takes less time when compared to the corresponding operations of sequential representation. Because of the advantages mentioned above, generally, linked representation is preferred for any data structure.

The advantages and disadvantages of the various data structures described in this book are tabulated below:

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known.	Slow search, slow deletion, and fixed size.
Ordered array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in, first-out access.	Slow access to other items.
Queue	Provides first-in, first-out access.	Slow access to other items.
Linked list	Quick insertion, quick deletion.	Slow search.
Binary tree	Quick search, insertion, deletion (if tree remains balanced)	Deletion algorithm is complex.
Red-black tree	Quick search, insertion, deletion. Tree always balanced.	Complex
Hash table	Very fast access if key known. Fast insertion.	Slow deletion, access slow if key not known, inefficient memory usage.
Heap	Fast insertion, deletion.	Slow access to other items.
Graph	Models real-world situations.	Some algorithms are slow and complex.

Abstract Data type (ADT)

An abstract data type (ADT) consists of a data type together with a set of operations, which define how the type may be manipulated. This specification is stated in an implementation independent manner.



Abstract data types exist conceptually and concentrate on the mathematical properties of the data type ignoring implementation constraints and details.

An abstract data type (ADT) is a specification of only the behavior of instances of that type. Such a specification may be all that is needed to design a module that uses the type. Primitive types are like ADTs. We know what the **int** type can do (add, subtract, multiply, etc.). But we need not know how it does these operations. And we need not even know how an **int** is actually stored. As clients, we can use the **int** operations without having to know how they are implemented. In fact, if we had to think about how they are implemented, it would probably be a distraction from designing the software that will use them.

The ADT uses generic terms for types: Integer instead of **int**, and Real instead of double. That is because it is supposed to be independent of any specific programming language. In general, a complete ADT would also include documentation that explains exactly how each operation should behave.

The advantages offered by abstract data types include:

- Modularity
- Precise specifications
- Information hiding
- Simplicity
- Integrity
- Implementation independence

Algorithm

An algorithm is a finite sequence of instructions for solving a stated problem. A stated problem is a well-defined task that has to be performed and must be solvable by an algorithm. The algorithm must eventually terminate, producing the required result.

Characteristics of an Algorithm

- **Input**: An algorithm should have zero or more inputs
- **Output**: An algorithm should have one or more outputs
- **Finiteness**: Every step in an algorithm should end in finite amount of time
- **Unambiguous**: Each step in an algorithm should clearly stated
- **Effectiveness**: Each step in an algorithm should be effective

To judge an algorithm the most important factors is to have a direct relationship to the performance of the algorithm. These have to do with their computing time & storage requirements (referred as Time complexity & Space complexity).

Example 1: Algorithm to test whether given entered number is odd or even.

Step 1: Start
Step 2: Read a
Step 3: Find modules of a by 2 (r = a % 2)
Step 4: If r= 0
    Print a is even
    Else
    Print a is odd
Step 5: stop

Example 2: Algorithm to find roots of quadratic equation.

Step 1: Start
Step 2: Input a, b, c
Step 3: calculate discriminant $d = b^2 - 4 \times a \times c$
Step 4: if d=0 then
    Print "Roots are equal"
    $root_1 = root_2 = \frac{-b}{2a}$
Step 5: else if d>0 then
    $root_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$
    $root_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$
Step 6: Else Print "Roots are imaginary"
Step 7: Print $Root_1$ and $Root_2$.
Step 8: Stop

Example 3: Algorithm to find largest number among given three numbers.

Step 1: Start
Step 2: Declare variables a, b, c
Step 3: Read values of a, b, c
Step 4: if a > b
    if a > c
        Print a is the greatest number
    else
        Print c is the greatest number
    Else
        if b > c
            Print b is the greatest number
        else
            Print c is the greatest number
Step 5: Stop

Exercise

1. What is main concept behind the asymptotic notations? Explain big oh in detail with suitable example.
2. What do you mean by ADT? Show that data type **int** as ADT.
3. What is main drawback of big oh notation?
4. What do you mean by linear data structure and non-linear data structure? Differentiate them with suitable example.
5. Solving a problem requires running an $O(N)$ algorithm and then afterwards a second $O(N)$algorithm. What is the total cost of solving the problem?
6. Solving a problem requires running an $O(N^2)$ algorithm and then afterwards an $O(N)$ algorithm. What is the total cost of solving the problem?
7. How can you convert a satisfiability problem into a three-satisfiability problem for an instance when an alternative in a Boolean expression has two variables? One variable?
8. Is it true that
   a. if f (n) is Θ(g(n)), then 2f (n) is Θ (2g(n))?
   b. f (n) + g(n) is Θ (min(f (n),g(n)))? c. 2na is O(2n)?
9. Group the following into equivalent Big-Oh functions:
   a. x^2, x, $x^2 + x$, $x^2 - x$, and $(x^3 / (x - 1))$.
10. What is RAM model? Describe their functionalities.
11. Define class P, NP and NP complete problems with suitable examples.
12. How can you determine space and time complexity of the algorithm? Describe with suitable example.
13. Write an algorithm to find roots of a quadratic equation.
14. Define worst, best and average case complexity of the algorithm. Explain with suitable example.
15. What is algorithm? Write down characteristics of algorithm.
16. Find big oh, big omega and big theta of following function,
    $F(x) = 3x^4+9x^2+8x+6$
17. List some well-known problems that are NP-complete when expressed as decision problems.
18. Define little oh, little omega with suitable example.
19. Why we need asymptotic notation? Explain.
20. Show that array as an ADT with suitable example.

•••

Chapter

3

Linked Lists

Arrays work well for unordered sequences and even for ordered sequences if they don't change much. But if we want to maintain an ordered list that allows quick insertions and deletions, we should use a linked data structure. The basic linked list consists of a collection of connected, dynamically allocated nodes.

Linked List

To overcome the disadvantage of fixed size arrays linked list were introduced.

A linked list consists of nodes of data which are connected with each other. Every node consists of two fields' data (info) and the link (next). The nodes are created dynamically.

- **Info:** the actual element to be stored in the list. It is also called data field.
- **Link:** one or two links that point to next and previous node in the list. It is also called next or pointer field.

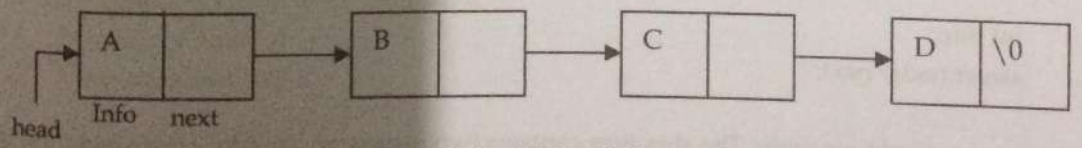


Fig: Singly linked list with four nodes.

What is the difference between Linked List and Linear Array?

S.No.	Array	Linked List
1.	Insertions and deletions are difficult.	Insertions and deletions can be done easily.
2.	It needs movements of elements for insertion and deletion	It does not need movement of nodes for insertion and deletion.
3.	In it space is wasted	In it space is not wasted.
4.	It is more expensive	It is less expensive.
5.	It requires less space as only information is stored	It requires more space as pointers are also stored along with information.
6.	Its size is fixed	Its size is not fixed.
7.	It cannot be extended or reduced according to requirements	It can be extended or reduced according to requirements.
8.	Same amount of time is required to access each element	Different amount of time is required to access each element.
9.	Elements are stored in consecutive memory locations.	Elements may or may not be stored in consecutive memory locations.
10.	If have to go to a particular element then we can reach their directly.	If we have to go to a particular node then we have to go through all those nodes that come before that node.

Self referential structure

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. Hence, a structure which contains a reference to itself is called self-referential structure. In general terms, this can be expressed as:

```
struct node
{
        member 1;
        member 2;
        ....................
        ....................
        struct node *name;
};
```

Example: Self referential structure

```
struct node
{
        int info;
        struct node *next;
};
```

This is a structure of type node. The structure contains two members: an info integer member, and a pointer to a structure of the same type (i.e., a pointer to a structure of type node), called next. Therefore this is a **self-referential** structure.

Header nodes

A header node is an extra node in a linked list that holds no data but serves to satisfy the requirement that every node containing an item have a previous node in the list. By using the header node, we greatly simplify the code—with a negligible space penalty. In more complex applications, header nodes not only simplify the code but also improve speed because, after all, fewer tests mean less time.

The use of a header node is somewhat controversial. Some argue that avoiding special cases is not sufficient justification for adding fictitious cells; they view the use of header nodes as little more than old-style hacking. Even so, we use them here precisely because they allow us to demonstrate the basic link manipulations without obscuring the code with special cases. Whether a header should be used is a matter of personal preference. Furthermore, in a class implementation, its use would be completely transparent to the user. However, we must be careful: The printing routine must skip over the header node, as must all searching routines. Moving to the front now means setting the current position to **header.next**, and so on.

Points to be noted for linked list

- The nodes in a linked list are not stored contiguously in the memory
- You don't have to shift any element in the list.
- Memory for each node can be allocated dynamically whenever the need arises.
- The size of a linked list can grow or shrink dynamically

Types of Linked Lists

There are four common types of Linked List:

1. Single linked list
2. Double linked list
3. Circular linked list
4. Circular doubly linked list

Valid operations on linked list

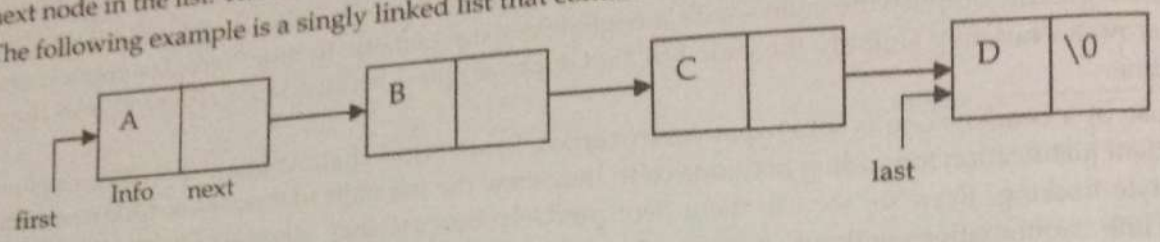
1. Inserting elements to linked list
   - Inserting an element at first position
   - Inserting an element at end
   - Inserting an element at specified position
   - Inserting an element from specified position
   - Inserting an element after given node
2. Deleting elements from linked list
   - Deleting an element at first position
   - Deleting an element at end
   - Deleting a node after given node
3. Searching a node
4. Merging linked lists

Singly linked list

A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made. In this type of linked list each node contains two fields

one is info field which is used to store the data items and another is link field that is used to point the next node in the list. The last node has a NULL pointer.

The following example is a singly linked list that contains four elements A, B, C and D.



Structure of a node of singly linked list

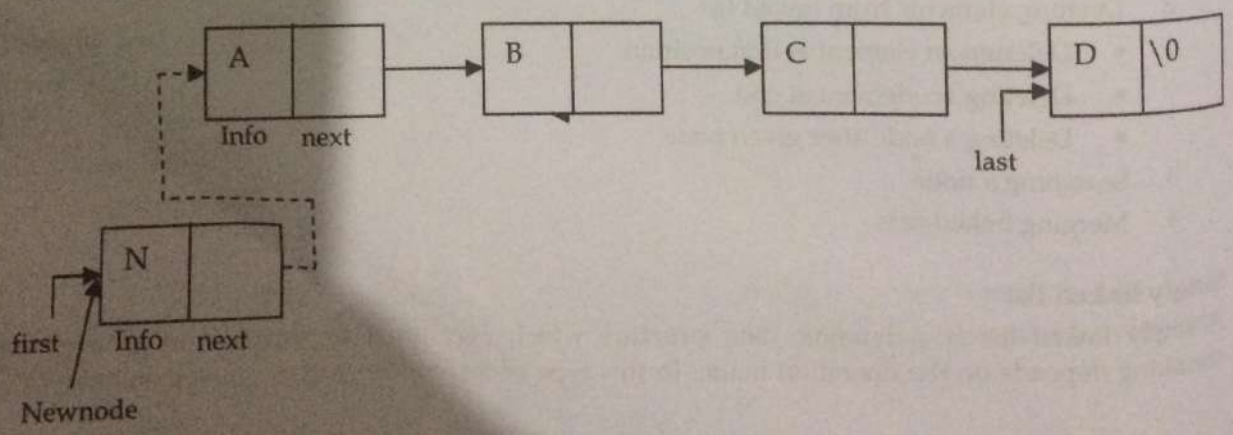
We can define a node as follows:

```
struct Node
{
        int info;
        struct Node *next;
};
typedef struct Node NodeType;
NodeType  *first;        //first is a pointer type structure variable that points to first node
NodeType  *last;  //last is a pointer type structure variable that points to last node
```

An algorithm to insert a node at the beginning of the singly linked list

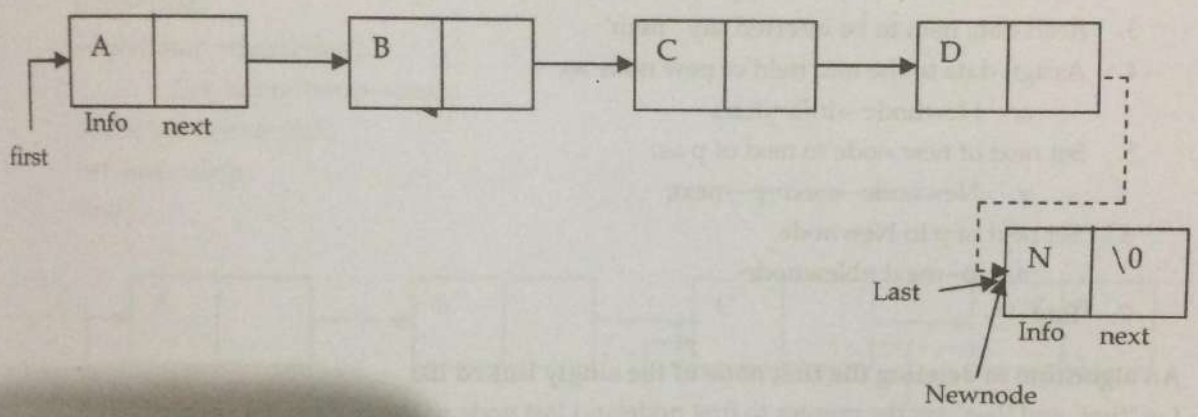
Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

1. Start
2. Create a new node using **malloc** function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say 'el'
4. Assign data to the info field of new node
   Newnode.info=el;
5. Set next of new node to first
   Newnode.next=first;
6. Set the first pointer to the new node
   first = Newnode;
7. End



An algorithm to insert a node at the end of the linked list:

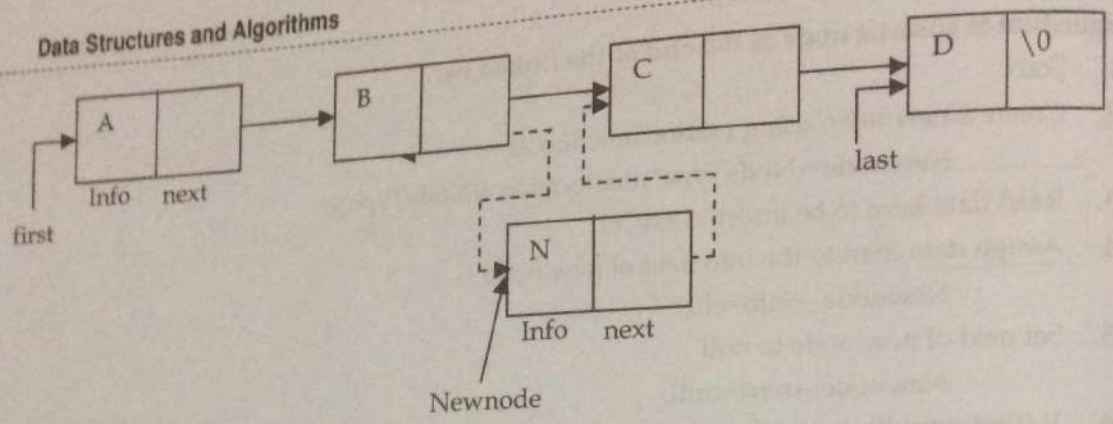
1. Start
2. Create a new node using **malloc** function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say 'el'
4. Assign data item to the info field of new node
   Newnode→info=el;
5. Set next of new node to null
   Newnode→next=null;
6. If (first ==null) then
   Set, first =last=Newnode and exit.
7. else
   Set, last→next=newnode;
   Set, last=newnode;
8. End



An algorithm to insert a node at the specified position in a singly linked list

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

1. Start
2. Create a new node using malloc function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Assign data to the info field of new node
   Newnode→info=el;
4. Enter position of a node at which you want to insert a new node. Let this position is 'pos'.
5. Set, temp=first;
6. if (first==null) then
   print "void insertion" and exit.
7. for(i=1; i<pos-1; i++)
   temp=temp→next;
8. Set, Newnode→next=temp→next;
9. Set, temp→next =Newnode.
10. End



An algorithm to insert a node after the given node in singly linked list

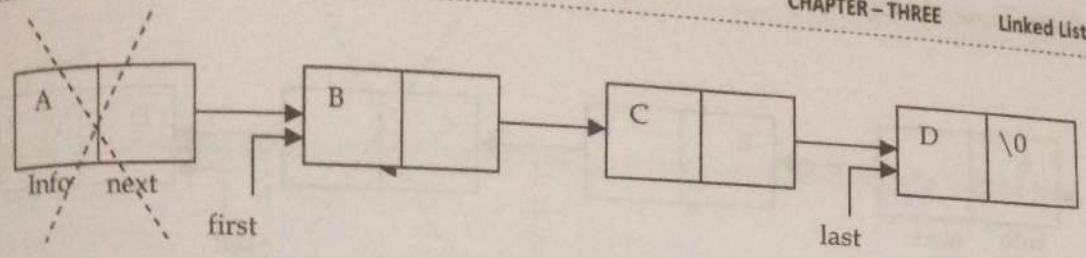
Let *first and *last be the pointer to first node and last node in the current list respectively and *p be the pointer to the node after which we want to insert a new node.

1. Start
2. Create a new node using **malloc** function
   a. Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say 'item'
4. Assign data to the info field of new node as,
   a. Newnode→info=item;
5. Set next of new node to next of p as,
   a. Newnode→next=p→next;
6. Set next of p to Newnode
   a. p→next =Newnode
7. End

An algorithm to deleting the first node of the singly linked list

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

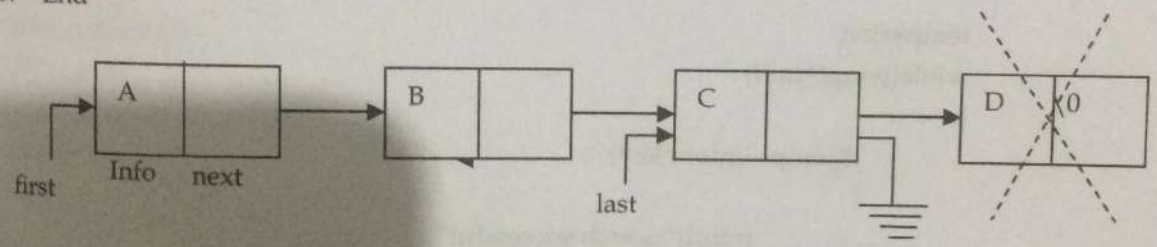
1. Start
2. If(first==null) then
   Print "Void deletion" and exit
3. Else if (first==last)
   Print deleted item as, first.info;
   first=last=null
4. Store the address of first node in a temporary variable **temp.**
   Set, temp=first;
5. Set first to next of first.
   Set, first=first→next;
6. Free the memory reserved by temp variable.
   free(temp);
7. End



An algorithm to deleting the last node of the singly linked list:

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

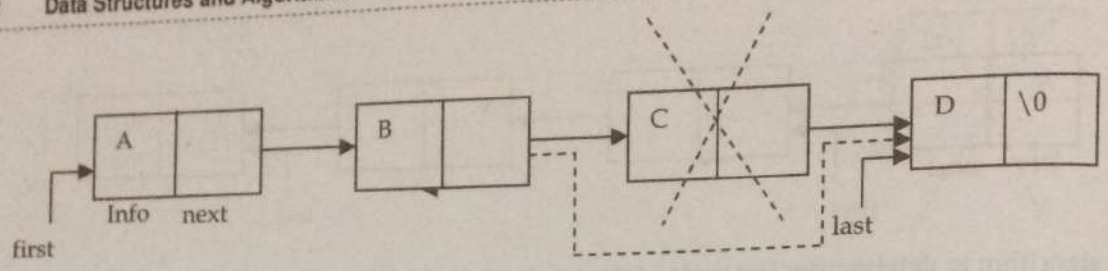
1. Start
2. If (first==null) then //if list is empty
   Print "Void deletion" and exit
3. else if (first==last) then //if list has only one node
   Print deleted item as, first.info;
   Set, first=last=null;
4. else
   temp=first;
   while(temp→next!=last)
   Set, temp=temp→next;
   Set, temp→next=null;
   Set, last=temp;
5. End



An algorithm to delete a node at the specified position in a singly linked list

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

1. Start
2. Read position of a node which to be deleted let it be 'pos'.
3. if first==null then,
   Print "void deletion" and exit
   Otherwise,
4. Enter position of a node at which you want to delete a new node. Let this position is 'pos'.
5. Set, temp=first
6. for(i=1; i<pos-1; i++)
   Set, temp=temp→next;
7. Print deleted item is temp.next.info
8. Set, loc=temp→next;
9. Set, temp→next =loc→next;
10. End



Searching an item in a linked list

To search an item from a given linked list we need to find the node that contain this data item. If we find such a node, then searching is successful otherwise searching unsuccessful.

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

```
void searchItem(int key)
{
        NodeType *temp;
        if(first==null)
        {
                print f("empty linked list");
                exit(0);
        }
        else
        {
                temp=first;
                while(temp!=null)
                {
                        if(temp→info==key)
                        {
                                printf("Search successful");
                                break;
                        }
                        temp=temp→next;
                }
                if(temp==null)
                        printf("Unsuccessful search");
        }
}
```

An algorithm to delete a node after the given node in singly linked list

Let *first and *last be the pointer to first node and last node in the current list and *p be the pointer to the node after which we want to delete a new node.

1. If (p==NULL or p→next==NULL) then
   a. print "deletion not possible and exit
2. Set loc=p→next
3. Set p→next=loc→next;

4. Free (loc)
5. End

Complete menu driven program in C to perform various operations in singly linked list

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>   //for malloc function
#include<process.h>  //for exit function
struct node
{
        int info;
        struct node *next;
};
typedef struct node NodeType;
NodeType  *first, *last;
first=last=NULL;
void insert_atfirst(int);
void insert_givenposition(int);
void insert_atend(int);
void delet_first();
void delet_last();
void delet_nthnode();
void info_sum();
void count_nodes();
void display();
void main( )
{
        int choice;
        int item;
        clrscr( );
        do
        {
                printf("\n manu for program:\n");
                printf("1. insert first \n2.insert at given position \n3 insert at last \n 4: Delete first
                node\n 5: delete last node\n6: delete nth node\n7: info sum\n8: count nodes\n9:
                Display items\n10:exit\n");
                printf("enter your choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                printf("Enter item to be inserted");
```

```
                    scanf("%d", &item)
                    insert_atfirst(item);
                    break;
            case 2:
                    printf("Enter item to be inserted");
                    scanf("%d", &item)
                    insert_givenposition(item);
                    break;
            case 3:
                    printf("Enter item to be inserted");
                    scanf("%d", &item)
                    insert_atend();
                    break;
            case 4:
                    delet_first();
                    break;
            case 5:
                    delet_last();
                    break;
            case 6:
                    delet_nthnode();
                    break;
            case 7:
                    info_sum();
                    break;
            case 8:
                    count_nodes();
                    break;
            case 9:
                    display();
                    break;
            case 10:
                    exit(1);
                    break;
            default:
                    printf("invalid choice\n");
            }
    }while(choice<10);
    getch();
}
void insert_atfirst(int item)
{
```

```
        NodeType *Newnode;
        Newnode=(NodeType*)malloc(sizeof(NodeType));
        if(first==null)
        {
                Newnode→next=null;
                first=newnode;
                last=newnode;
        }
        else
        {
                Newnode→next=first;
                first=Newnode;
        }
}
void insert_givenposition(int item)
{
        int pos, i;
        NodeType *Newnode, *temp;
        newnode→info=item;
        printf(" Enter position of a node at which you want to insert a new node");
        scanf("%d", &pos);
        if(first==null)
        {
                first=newnode;
                last=newnode;
        }
        else
        {
                temp=first;
                for(i=1; i<pos-1; i++)
                {
                        temp=temp→next;
                }
                Newnode→next=temp→next;
                temp→next =Newnode.
        }
}
void insert_atend(int item)
{
        NodeType *NewNode;
        Newnode=(NodeType*)malloc(sizeof(NodeType));
        Newnode→info=item;
```

```
        Newnode→next=NULL;
        if(first==null)
        {
            first=newnode;
            last=newnode;
        }
        else
        {
            last→next=newnode;
            last=newnode;
        }
}

void delet_first()
{
        NodeType *temp;
        if(head==NULL)
        {
            printf("Void deletion|n");
            return;
        }
        else
        {
            temp=head;
            head=head->next;
            free(temp);
        }
}
void delet_last()
{
        NodeType *hold,*temp;
        if(head==NULL)
        {
            printf("Void deletion|n");
            return;
        }
        else if(head→next==NULL)
        {
            hold=head;
            head=NULL;
            free(hold);
        }
```

```
        else
        {
                temp=head;
                while(temp→next→>next!=NULL)
                        temp=temp→next;
                hold=temp→next;
                temp→next=NULL;
                free(hold);
        }
}
void delet_nthnode()
{
        NodeType  *hold,*temp;
        int pos, i;
        if(first==NULL)
        {
                printf("Void deletion\n");
                return;
        }
        else
        {
                temp=first;
                printf("Enter position of node which node is to be deleted\n");
                scanf("%d",&pos);
                for(i=1;i<pos-1;i++)
                        temp=temp→next;
                hold=temp→next;
                temp→next=hold→next;
                free(hold);
        }
}
void info_sum( )
{
        NodeType *temp;
        temp=first;
        while(temp!=NULL)
        {
                printf("%d\t", temp→info);
                temp=temp→next;
        }
}
void count_nodes()
```

```
{
        int cnt=0;
        NodeType *temp;
        temp=first;
        while(temp!=NULL)
        {
                cnt++;
                temp=temp→next;
        }
        printf("total nodes=%d", cnt);
        }
}
void display()
{
        NodeType *temp;
        temp=first;
        if(first==null)
        {
                printf("Empty linked list");
                exit(1);
        }
        else
        {
                while(temp!=NULL)
                {
                        printf(" %d\t",temp→info);
                        temp=temp→next;
                }
        }
}
```

Output

Menu for program
1. insert first
2. insert at given position
3: insert at last
4: Delete first node
5: delete last node
6: delete nth node
7: info sum
8: count nodes
9: Display items

10: exit
Enter your choice
1
Enter data item to be inserted
43
Enter your choice
1
Enter data item to be inserted
12
Enter your choice
1
Enter data item to be inserted
88
Enter your choice
9
88 12 43 Enter your choice
4
12 43 Enter your choice
10

Circular Linked list

A circular linked list is a list where the link field of last node points to the first node of the list. Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node. The main advantage of circular linked list is that it requires minimum time to traverse the nodes which are already traversed, without moving to starting node.

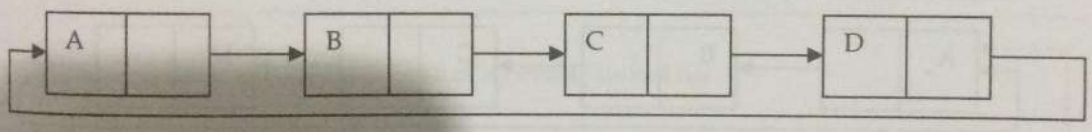


Fig: Circular linked list

Advantages and disadvantages of circular linked list

Advantages

1. If we are at a node, then we can go to any node. But in linear linked list it is not possible to go to previous node
2. It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes. But in double linked list, we will have to go through in between nodes

Disadvantages

1. It is not easy to reverse the linked list
2. If proper care is not taken, then the problem of infinite loop can occur
3. If we at a node and go back to the previous node, then we cannot do it in single step.

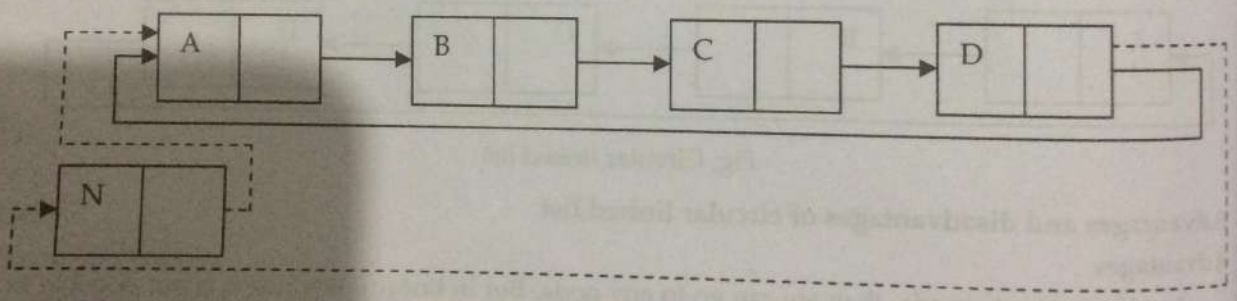
Representation of circular linked list

We declare the structure for the circular linked list in the same way as declared it for the singly linked list.

```
struct Node
{
        int info;
        struct Node *next;
};
typedef struct Node NodeType;
NodeType  *first;
NodeType  *last;
```

Algorithm to insert a node at the beginning of a circular linked list

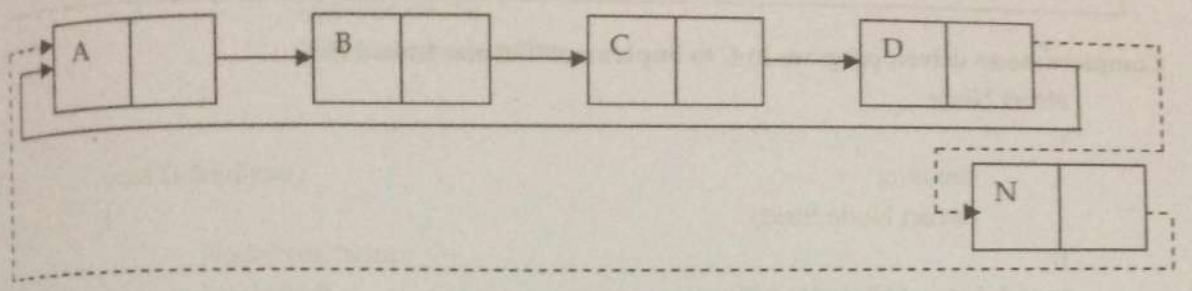
1. Create a new node by using malloc function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType));
2. Read data item to be inserted say it be 'el'
3. Set Newnode→info=el
4. if first==null then
   Set, Newnode→next=Newnode
   Set, first=Newnode
   Set, last =Newnode
5. else
   Set, Newnode→next=start
   Set, first=Newnode
   Set, last→next=Newnode
6. End



Algorithm to insert a node at the end of a circular linked list

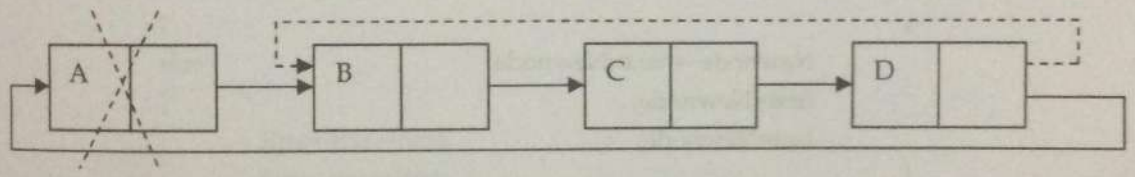
1. Create a new node by using malloc function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType));
2. Read data item to be inserted say it be 'el'
3. Set Newnode→info=el
4. if start==null then
   Set, Newnode→next=Newnode
   Set, start=Newnode
   Set, last Newnode

5. else
   Set, last.next=Newnode
   Set, last=Newnode
   Set, last.next=start
6. End



Algorithm to delete a node from the beginning of a circular linked list

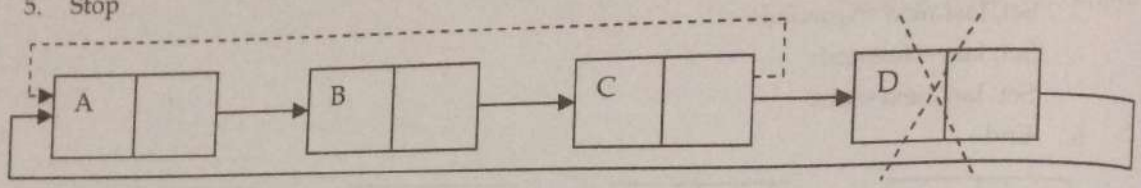
1. Start
2. if first==null then
   Print "Empty list" and exit
3. else
   Print the deleted element=first→info
   Set, first=first→next
   Set, last→next=first;
4. Stop



Algorithm to delete a node from the end of a circular linked list

1. Start
2. if first==null then
   Print "empty list" and exit
3. else if first==last
   - Print deleted element=first.info
   - Set, first=last=null
4. else
   Set, temp=first
   while(temp→next!=last)
   Set, temp=temp→next
   End while
   Print the deleted element=last→info
   Set, last=temp
   Set, last→next=first

5. Stop



Complete menu driven program in C to implement Circular linked list

```
struct Node
{
        int info;
        struct Node *next;
};
typedef struct Node NodeType;
NodeType  *first;
NodeType  *last;
first=null;
last=null;
void insertbeg(int item)
{
        Newnode=(NodeType*)malloc(sizeof(NodeType));
        Newnode→info=item;
        if(first==null)
        {
                Newnode→next=Newnode;
                first=Newnode;
                last=Newnode;
        }
        else
        {
                Newnode→next=first;
                first=Newnode;
                last→next=Newnode;
        }
}
void insertEnd(int item)
{
        Newnode=(NodeType*)malloc(sizeof(NodeType));
        Newnode→info=item;
        if(first==null)
        {
                first=Newnode;
                last=Newnode;
```

```
                Newnode→next=Newnode;
        }
        else
        {
                last→next=Newnode;
                last=Newnode;
                Newnode→next=first;
        }
}
void DeleteFirst()
{
        NodeType *temp;
        temp=first;
        if(first==null)
        {
                printf("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
                free(temp);
        }
        else
        {
                first=first→next;
                last→next=first;
                free(temp);
        }
}
void DeleteLast()
{
        NodeType *temp;
        temp=last;
        if(last==null)
        {
                printf("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
```

```
                free(temp);
        }
        else
        {
                while(temp→next!=last)
                {
                        temp=temp→next;
                }
                temp→next=first;
                last=temp;
                temp=first;
                free(temp);
        }
}
void Display()
{
        NodeType *temp;
        if(first==null)
        {
                printf("Empty linked list");
        }
        else
        {
                temp=first;
                while(temp!=last)
                {
                        printf(temp→info);
                        temp=temp→next;
                }
                prinf(last→info);
        }
}
void main()
{
        int choice;
        int item;
        printf("1:Insert at beginning");
        printf("2:Insert at last");
        printf("3:delete first node");
        printf("4:delete last node");
        printf("5:Display");
        do
```

```
    {
        printf("Enter your choice");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter data item to be inserted");
                scanf("%d", &item);
                insertbeg(item);
                break;
            case 2:
                printf("Enter data item to be inserted");
                scanf("%d", &item);
                insertEnd(item);
                break;
            case 3:
                DeleteFirst();
                break;
            case 4:
                DeleteLast();
                break;
            case 5:
                Display();
                break;
            default:
                printf("Invalid choice Please enter correct choice");
            }
        }while(choice<6);
    }
}
```

Output

```
1: Insert at beginning
2: Insert at last
3: delete first node
4: delete last node
5: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
```

```
Enter data item to be inserted
87
Enter your choice
1
Enter data item to be inserted
32
Enter your choice
5
32 87 22 Enter your choice
2
Enter data item to be inserted
908
Enter your choice
5
32 87 22 908 Enter your choice
3
Enter your choice
5
87 22 908 Enter your choice
4
Enter your choice
5
87 22 Enter your choice
3
Enter your choice
5
22 Enter your choice
3
Enter your choice
5
Empty linked list
Enter your choice
```

Doubly Linked List (DLL)

A linked list in which all nodes are linked together by multiple numbers of links i.e. each node contains three fields (two pointer fields and one data field) rather is called doubly linked list. It provides bidirectional traversal. Doubly circular linked list can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.

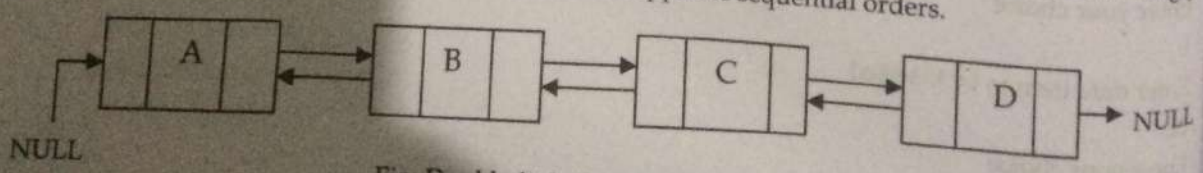


Fig: Doubly linked list with three nodes

The above diagram represents the basic structure of Doubly Circular Linked List. In doubly circular linked list, the previous link of the first node points to the last node and the next link of the last node points to the first node. In doubly circular linked list, each node contains two fields called links used to represent references to the previous and the next node in the sequence of nodes

Advantages and disadvantages of doubly linked list

Advantages

1. We can traverse in both directions i.e. from starting to end and as well as from end to starting
2. It is easy to reverse the linked list
3. If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node

Disadvantages

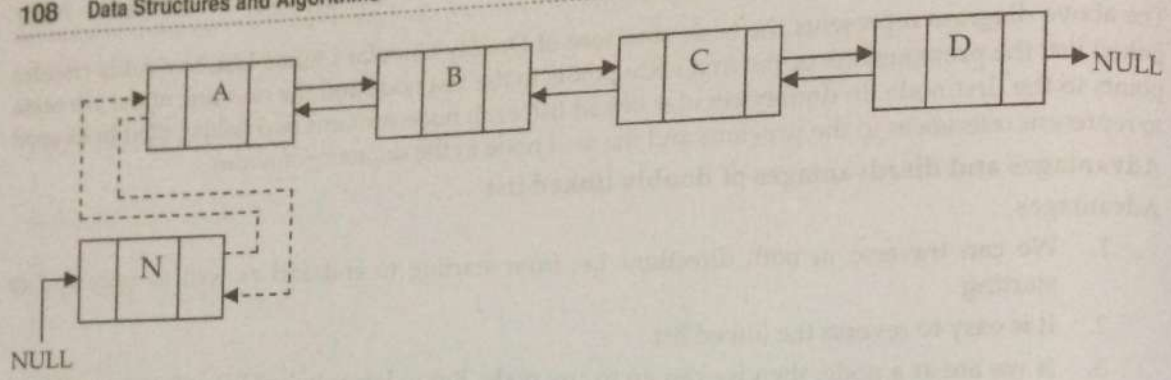
1. It requires more space per space per node because one extra field is required for pointer to previous node
2. Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list.

Representation of doubly linked list

```
struct node
{
        int info;
        struct node *prev;
        struct node *next;
};
typedef struct node NodeType;
NodeType *first=NULL;
NodeType *last=NULL;
```

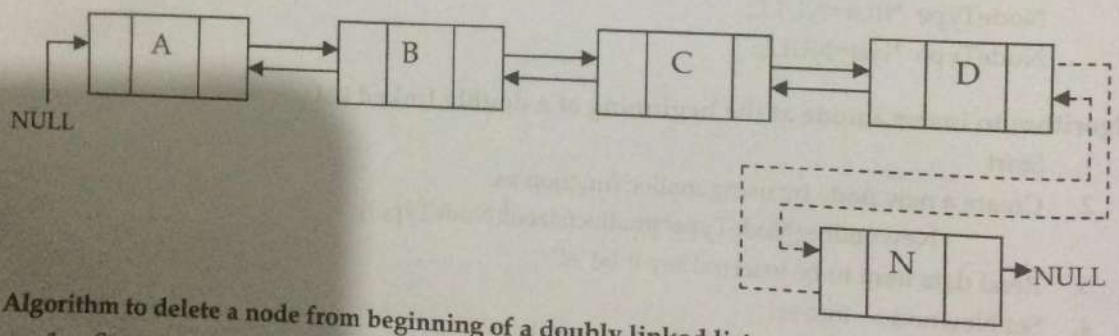
Algorithm to insert a node at the beginning of a doubly linked list

1. Start
2. Create a new node by using malloc function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType))
3. Read data item to be inserted say it be 'el'
4. Set Newnode→info=el
5. Set Newnode→prev=Newnode→next=null
6. If first==null then
   Set, first=last=Newnode
   Otherwise,
7. Set Newnode→next=first
8. Set first→prev=Newnode
9. Set first=Newnode
10. Stop



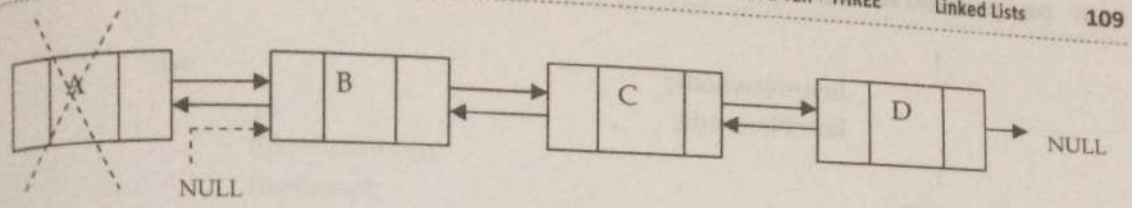
Algorithm to insert a node at the end of a doubly linked list

1. Start
2. Create a new node by using malloc function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType))
3. Read data item to be inserted say it be 'el'
4. Set Newnode→info=el
5. Set Newnode→next=NULL
6. If first==NULL then
   Set, first=last=Newnode
   Otherwise,
7. Set last→next=Newnode
8. Set Newnode→prev=last
9. Set last=Newnode
10. stop



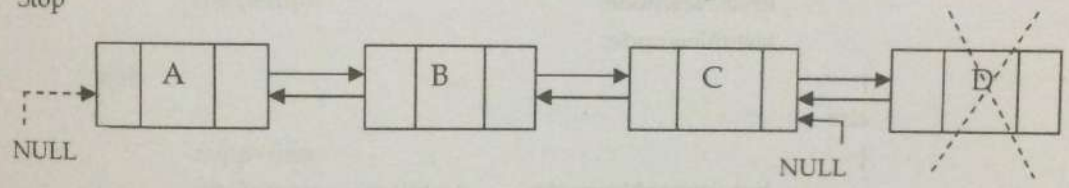
Algorithm to delete a node from beginning of a doubly linked list

1. Start
2. if first==NULL then
   Print "empty list" and exit
3. else
   Set, temp=first
   Set, first=first→next
   Set, first→prev=null
   Free(temp)
4. Stop



Algorithm to delete a node from end of a doubly linked list

1. Start
2. if first==NULL then
   Print "empty list" and exit
3. else if(first==last) then
   Set, first=last=NULL
4. else
   Set, temp=first;
   while(temp→next!=last)
   temp=temp→next
   end while
   Set temp→next=null
   Set, last=temp
5. Stop



Complete menu driven program in C to implement doubly linked list

```
struct node
{
        int info;
        struct node *prev;
        struct node *next;
};
typedef struct node NodeType;
NodeType  *first=NULL;
NodeType  *last=NULL;
void insertbeg(int el)
{
        NodeType *Newnode;
        Newnode=(NodeType*)malloc(sizeof(NodeType));
        Newnode→info=el;
        Newnode→prev=Newnode→next=null;
        if(first==null)
```

```
        {
                first=Newnode;
                last=Newnode;
        }
        else
        {
                Newnode→next=first;
                first→prev=Newnode;
                first=Newnode;
        }
}
void insertEnd(int el)
{
        NodeType *Newnode;
        Newnode=(NodeType*)malloc(sizeof(NodeType));
        Newnode→info=el;
        Newnode→prev=Newnode→next=null;
        if(first==null)
        {
                first=Newnode;
                last=Newnode;
        }
        else
        {
                last→next=Newnode;
                Newnode→prev=last;
                last=Newnode;
        }
}
void DeleteFirst()
{
        NodeType *temp;
        temp=first;
        if(first==null)
        {
                printf("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
                free(temp);
```

```
        }
        else
        {
                first=first→next;
                free(temp);
        }
}
void DeleteLast()
{
        NodeType *temp, *hold;
        temp=first;
        if(last==null)
        {
                printf("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
                free(temp);
        }
        else
        {
                temp=first;
                while(temp→next!=last)
                {
                        temp=temp→next;
                }
                hold=temp→next;
                temp→next=null;
                last=temp;
                free(hold);
        }
}

void Display()
{
        NodeType *temp;
        temp=first;
        if(first==null)
        {
                printf("Empty linked list");
```

```
        }
        else
        {
                while(temp!=last)
                {
                        printf(temp→info);
                        temp=temp→next;
                }
                printf(last→info);
        }
}
void main()
{
        int choice;
        int item;
        printf("1:Insert at beginning");
        printf("2:Insert at last");
        printf("3:delete first node");
        printf("4:delete last node");
        printf("5:Display");
        do
        {
                printf("Enter your choice");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                printf("Enter data item to be inserted");
                                scanf("%d", &item);
                                insertbeg(item);
                                break;
                        case 2:
                                printf("Enter data item to be inserted");
                                scanf("%d", &item);
                                insertEnd(item);
                                break;
                        case 3:
                                DeleteFirst();
                                break;
                        case 4:
                                DeleteLast();
                                break;
```

```
        }
}
```

Output

```
1: Insert at begin
2: Insert at last
3: delete first no
4: delete last nod
5: Display
Enter your choic
1
Enter data item t
33
Enter your choic
2
Enter data item t
77
Enter your choic
1
Enter data item t
98
Enter your choic
5
98 33 77 Enter y
2
Enter data item t
54
Enter your choic
5
98 33 77 54 Ent
2
Enter data item t
5
Enter your choic
5
98 33 77 54 5 E
```

```
                case 5:
                        Display();
                        break;
                default:
                        printf("Invalid choice Plz enter correct choice");
            }
        }while(choice<6);
    }
}
```

Output

```
1: Insert at beginning
2: Insert at last
3: delete first node
4: delete last node
5: Display
Enter your choice
1
Enter data item to be inserted
33
Enter your choice
2
Enter data item to be inserted
77
Enter your choice
1
Enter data item to be inserted
98
Enter your choice
5
98 33 77 Enter your choice
2
Enter data item to be inserted
54
Enter your choice
5
98 33 77 54 Enter your choice
2
Enter data item to be inserted
5
Enter your choice
5
98 33 77 54 5 Enter your choice
```

3
Enter your choice
5
33 77 54 5 Enter your choice
4
Enter your choice
5
33 77 54 Enter your choice
4
Enter your choice
5

Circular Doubly Linked List

A circular doubly linked list is one which has he successor and predecessor pointer in circular manner. It is a doubly linked list where the next link of last node points to the first node and previous link of first node points to last node of the list. The main objective of considering circular doubly linked list is to simplify the insertion and deletion operations performed on doubly linked list.

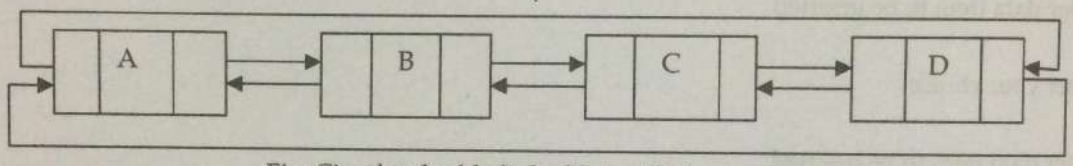


Fig: Circular doubly linked list with four nodes

Representation of circular doubly linked list

```
struct node
{
        int info;
        struct node *prev;
        struct node *next;
};
typedef struct node NodeType;
NodeType *first=NULL;
NodeType *last=NULL;
```

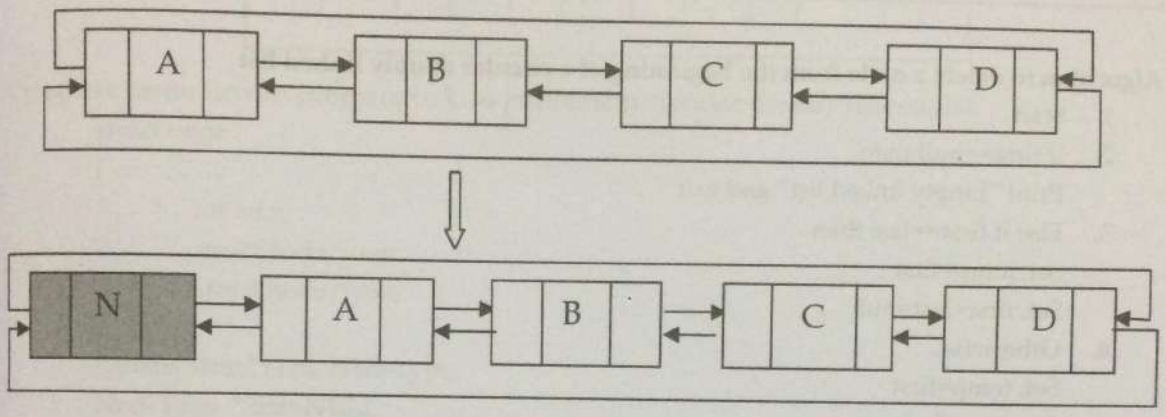
Algorithm to insert a node at the beginning of a circular doubly linked list

1. Start
2. Create a new node by using malloc function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say it be 'el'
4. Set Newnode→info=el
5. If first==null then
   Set, first=last=Newnode

Set, Newnode→next=Newnode
Set, Newnode→prev=Newnode

Otherwise,

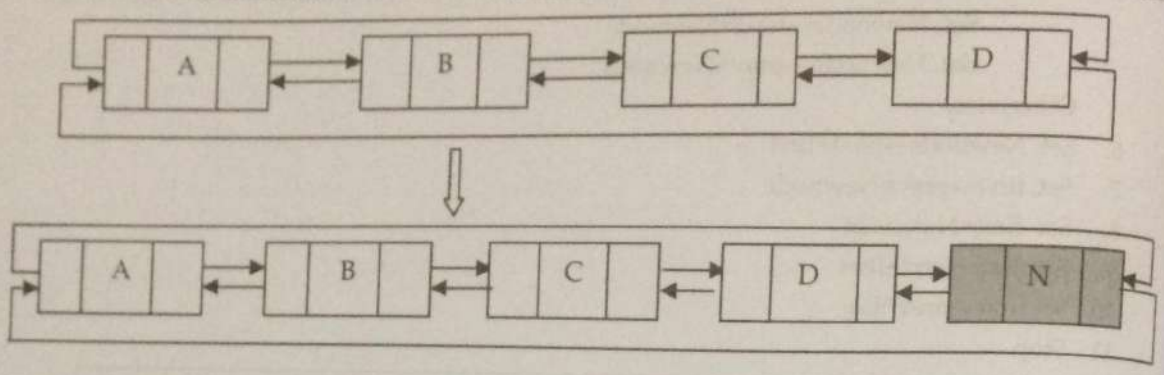
6. Set Newnode→next=first
7. Set, first→prev=Newnode
8. Set first=Newnode
9. Set last→next=first
10. Set first→prev=last
11. Stop



Algorithm to insert a node at the end of a circular doubly linked list

1. Start
2. Create a new node by using malloc function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say it be 'el'
4. Set Newnode→info=el
5. If first==null then
   Set, first=last=Newnode
   Set, Newnode→next=Newnode
   Set, Newnode→prev=Newnode

   Otherwise,
6. Newnode→next=first
7. Set, first→prev=Newnode
8. Set last→next=Newnode
9. Set Newnode→prev=last
10. Set last=Newnode
11. Stop

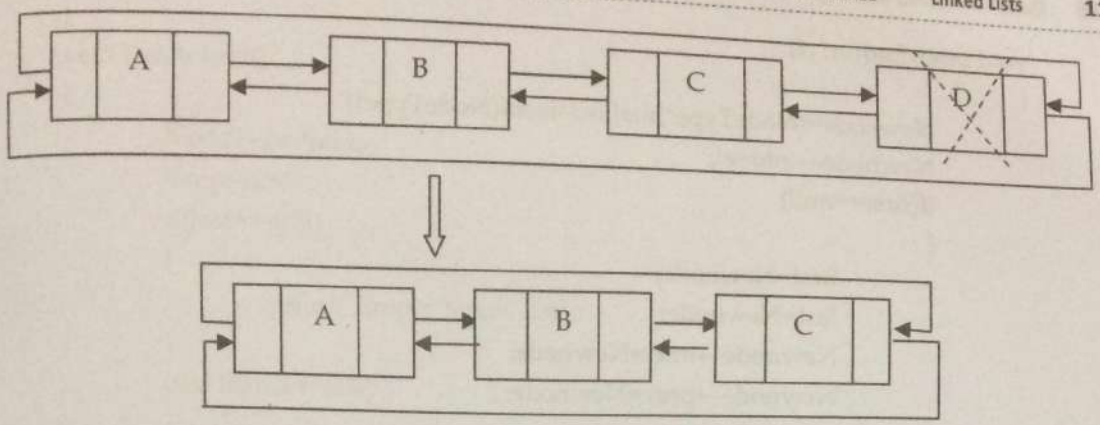


Algorithm to delete a node from the beginning of a circular doubly linked list

1. Start
2. If first==null then
   Print "Empty linked list" and exit
3. Else if first==last then
   Set, temp=first
   Set, first=last=null
4. Otherwise,
   Set, temp=first
   Set, first=first→next
   Set, first→prev=last
   Set, last→next=first
5. Free(temp)
6. Stop

Algorithm to delete a node from the end of a circular doubly linked list

1. Start
2. If first==null then
   Print "Empty linked list" and exit
3. Else if first==last then
   Set, temp=first
   Set, first=last=null
4. Otherwise,
   Set, temp=first
   while(temp→next!=last)
   Set, temp=temp→next
   End while
   Set, last=temp
   Set, last→next=first
   Set, first→prev=last
5. Stop



Complete menu driven program in C to implement circular doubly linked list

```
struct node
{
        int info;
        struct node *prev;
        struct node *next;
};
typedef struct node NodeType;
NodeType  *first=NULL;
NodeType  *last=NULL;

void insertbeg(int el)
{
        Newnode=(NodeType*)malloc(sizeof(NodeType))
        Newnode→info=el;
        if(first==null)
        {
                first=Newnode;
                last=Newnode;
                Newnode→next=Newnode;
                Newnode→prev=Newnode;
        }
        else
        {
                Newnode→next=first;
                first→prev=Newnode;
                first=Newnode;
                last→next=first;
                first→prev=last;
        }
}
```

```
void insertEnd(int el)
{
        Newnode=(NodeType*)malloc(sizeof(NodeType))
        Newnode→info=el;
        if(first==null)
        {
                first=Newnode;
                last=Newnode;
                Newnode→next=Newnode;
                Newnode→prev=Newnode;
        }
        else
        {
                last→next=Newnode;
                Newnode→prev=last;
                last=Newnode;
                last→next=first;
                first→prev=last;
        }
}

void DeleteFirst()
{
        NodeType *temp;
        temp=first;
        if(first==null)
        {
                printf("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
                free(temp);
        }
        else
        {
                first=first→next;
                last→next=first;
                first→prev=last;
                free(temp);
        }
```

```
}
void DeleteLast()
{
        NodeType *temp;
        temp=first;
        if(last==null)
        {
                printf("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
                free(temp);
        }
        else
        {
                while(temp→next!=last)
                {
                        temp=temp→next;
                }
                last=temp;
                last→next=first;
                first→prev=last;
                free(temp→next);
        }
}

void Display()
{
        NodeType *temp;
        temp=first;
        if(first==null)
        {
                printf("Empty linked list");
        }
        else
        {
                while(temp!=last)
                {
                        printf("%d\t", temp→info);
                        temp=temp→next;
```

```
                }
                printf("%d", last→info);
        }
}

void main()
{
        int choice;
        int item;
        printf("1:Insert at beginning");
        printf("2:Insert at last");
        printf("3:delete first node");
        printf("4:delete last node");
        printf("5:Display");
        do
        {
                printf("Enter your choice");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                printf("Enter data item to be inserted");
                                scanf("%d", &item);
                                insertbeg(item);
                                break;
                        case 2:
                                printf("Enter data item to be inserted");
                                scanf("%d", &item);
                                insertEnd(item);
                                break;
                        case 3:
                                DeleteFirst();
                                break;
                        case 4:
                                DeleteLast();
                                break;
                        case 5:
                                Display();
                                break;
                        default:
                                printf("Invalid choice Plz enter correct choice");
                }
```

Output

```
1: Insert a
2: Insert a
3: delete fi
4: delete la
5: Display
Enter your
1
Enter data
22
Enter your
1
Enter data i
99
Enter your c
5
99 22 Enter
2
Enter data ite
16
Enter your ch
5
99 22 16 Ente
3
Enter your cho
5
22 16 Enter yo
4
Enter your cho
5
22 Enter your c
3
Enter your choi
5
Empty linked lis
Enter your choic
```

```
        }while(choice<6);
    }
}
```

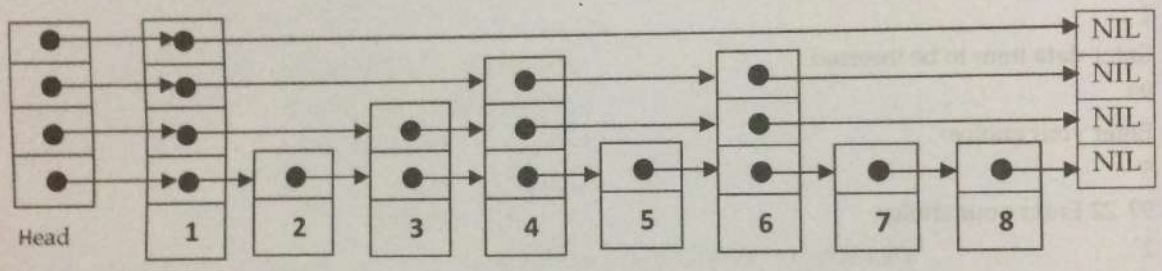
Output

```
1: Insert at beginning
2: Insert at last
3: delete first node
4: delete last node
5: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
99
Enter your choice
5
99 22 Enter your choice
2
Enter data item to be inserted
16
Enter your choice
5
99 22 16 Enter your choice
3
Enter your choice
5
22 16 Enter your choice
4
Enter your choice
5
22 Enter your choice
3
Enter your choice
5
Empty linked list
Enter your choice
```

Skip Lists

Linked lists have one serious drawback: They require sequential scanning to locate a searched-for element. The search starts from the beginning of the list and stops when either a searched-for element is found or the end of the list is reached without finding this element. Ordering elements on the list can speed up searching, but a sequential search is still required. Therefore, we may think about lists that allow for skipping certain nodes to avoid sequential processing. A **skip list** is an interesting variant of the ordered linked list that makes such a non sequential search possible.

In computer science, a skip list is a data structure that allows fast search within an ordered sequence of elements. Fast search is made possible by maintaining a linked hierarchy of subsequences, each skipping over fewer elements. Searching starts in the sparsest subsequence until two consecutive elements have been found, one smaller and one larger than or equal to the element searched for. Via the linked hierarchy, these two elements link to elements of the next sparsest subsequence, where searching is continued until finally we are searching in the full sequence. The elements that are skipped over may be chosen probabilistically or deterministically, with the former being more common.



The above fig shows a schematic picture of the skip list data structure. Each box with an arrow represents a pointer and a row is a linked list giving a sparse subsequence; the numbered boxes at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the top until consecutive elements bracketing the search element are found.

A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer i+1 with some fixed probability p (two commonly used values for p are 1/2 or 1/4). On average, each element appears in 1/(1-p) lists, and the tallest element (usually a special head element at the front of the skip list) in $\left(\log \frac{1}{p^n}\right)$ lists.

A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, or the search reaches the end of the linked list, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps in each linked list is at most 1/p, which can be seen by tracing the search path backwards from the target until reaching an element that appears in the next higher list or reaching the beginning of the current list. Therefore, the total expected cost of a search is $\frac{\left(\log \frac{1}{p^n}\right)}{p}$ which is O(log n) when p is a constant. By choosing different values of p, it is possible to trade search costs against storage costs.

Algorithm

```
Find (element el)
    p = the non-null list on the highest level i;
    while el not found and i ≥ 0
        if p.key < el
            p = a sub-list that begins in the predecessor of p on level --i;
        else if p.key > el
            if p is the last node on level i
                p = a non-null sub-list that begins in p on the highest level < i;
                i = the number of the new level;
            else p = p.next;
```

Structure of skip list

```
struct SkipListEntry
{
    char key[20];
    int value;
    struct SkipListEntry up;     // up link
    struct SkipListEntry down;   // down link
    struct SkipListEntry left;   // left link
    struct SkipListEntry right;  // right link
    ...........
}
```

Exercise

1. What is list? Differentiate between list and linked list.
2. What is main advantage and disadvantage of using doubly linked list over singly linked list?
3. Write an algorithm for printing a singly linked list in reverse, using only constant extra space. This instruction implies that you cannot use recursion but you may assume that your algorithm is a list method.
4. Suggest an array implementation of linked lists.
5. What is main concept behind skip list?
6. Write a method to check whether two singly linked lists have the same contents.
7. Write a method to reverse a singly linked list using only one pass through the list.
8. Put numbers in a singly linked list in ascending order. Use this operation to find the median in the list of numbers.
9. How can a singly linked list be implemented so that insertion requires no test for whether head is null?
10. One way to implement a queue is to use a circularly linked list. Assume that the list does not contain a header and that you can maintain one iterator for the list. For which of the following representations can all basic queue operations be performed in constant worst-case time? Justify your answers.

a. Maintain an iterator that corresponds to the first item in the list.
b. Maintain an iterator that corresponds to the last item in the list.

11. What does the following function do for a given Linked List?

```
void fun1(struct Node* head)
{
        if(head == NULL)
                return;
        fun1(head->next);
        printf("%d ", head->data);
}
```

12. What type of memory allocation is referred for Linked lists?
13. Describe what is Node in link list? And name the types of Linked Lists?
14. Mention what are the applications of Linked Lists?
15. What does the dummy header in linked list contain?
16. Mention what is the difference between singly and doubly linked lists?
17. Mention how to insert a new node in linked list where free node will be available?
18. Mention for which header list, you will found the last node contains the null pointer?
19. Mention how to delete first node from singly linked list?
20. Mention the steps to insert data at the starting of a singly linked list?

•••

Chapter

4

Stacks and Its Applications

What is stack?

A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.Stack uses a variable called top which points topmost element in the stack. top is incremented while **pushing** (inserting) an element in to the stack and decremented while **popping** (deleting) an element from the stack.

A stack follows the principle of last-in-first-out (LIFO) system. According to the stack terminology, PUSH and POP are two terms used for insert and delete operations.

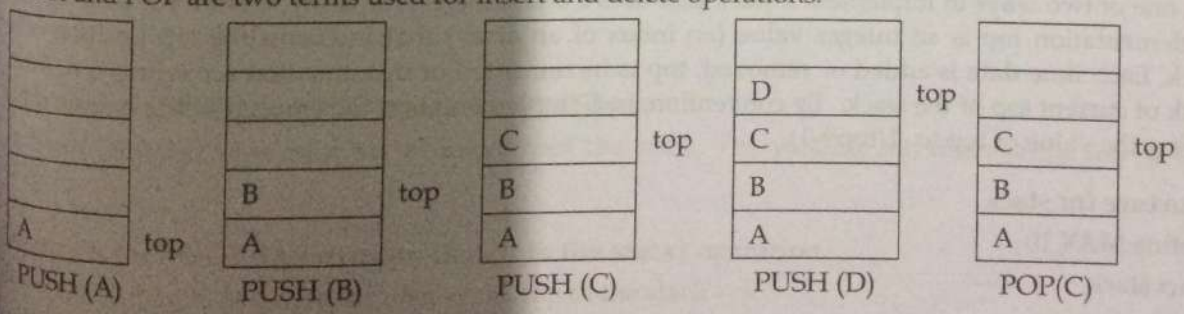


Fig: push, pop operations in Stack

Representation of Stacks

A stack may be represented by means of a one-way list or a linear array. Unless, otherwise stated, each of the stacks will be maintained by a linear array STACK; a variable TOP contains the location of the top element of the stack. A variable N gives the maximum number elements that can be held by the stack. The condition where TOP is NULL, indicate that the stack is empty. The condition where TOP is N, will indicate that the stack is full.

Applications of Stack

Stack is used directly and indirectly in the following fields:

- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser
- To perform the undo sequence in a text editor
- Used in recursion
- To check the correctness of parentheses sequence
- To pass the parameters between the functions in a C program
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures

The Stack ADT

A **stack** of elements of type *T* is a finite sequence of elements of *T* together with the operations:

- **CreateEmptyStack(S):** Create or make stack *S* be an empty stack
- **Push(S, x):** Insert *x* at one end of the stack, called its **top**
- **Top(S):** If stack *S* is not empty; then retrieve the element at its **top**
- **Pop(S):** If stack *S* is not empty; then delete the element at its **top**
- **IsFull(S):** Determine whether stack *S* is full or not. Return **true** if *S* is full; return **false** otherwise.
- **IsEmpty(S):** Determine whether stack *S* is empty or not. Return **true** if *S* is an empty stack; return **false** otherwise.

Implementation of Stack

Stack can be implemented in following two ways:

1. Array Implementation of stack (or static implementation)
2. Linked list implementation of stack (or dynamic)

Array (static) implementation of a stack

It is one of two ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value (an index of an array) that indicates the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack. By convention, in C implementation the empty stack is indicated by setting the value of top to -1(top=-1).

Structure for stack

```
#define MAX 10
struct stack
```

```
{
        int items[MAX];          //Declaring an array to store items
        int top;                 //Top of a stack
};
typedef struct stack st;
```

Creating Empty stack

The value of top=-1 indicates the empty stack in C implementation.

```
void create_empty_stack(struct stack e)   /*Function to create an empty stack*/
{
        e.top=-1;
}
```

Stack Empty or Underflow

This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty. The following function return 1 if the stack is empty, 0 otherwise.

```
int IsEmpty()
{
        if(top==-1)
                return 1;
        else
                return 0;
}
```

Stack Full or Overflow

This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location (MAXSIZE- 1) of the stack. The following function returns true (1) if stack is full false (0) otherwise.

```
int IsFull()
{
        if(top==MAXSIZE-1)
                return 1;
        else
                return 0;
}
```

Algorithm for PUSH and POP operations on Stack

Let Stack [MAXSIZE] is an array to implement the stack. The variable top denotes the top of the stack.

Algorithm for PUSH (inserting an item into the stack) operation

This algorithm adds or inserts an item at the top of the stack

1. Check for stack overflow as
   if top==MAXSIZE-1 then

print "Stack Overflow" and Exit the program

else

Increase top by 1 as

Set, top=top+1

2. Read elements to be inserted say element
3. Set, Stack[top]= element //Inserts item in new top position
4. Stop

Algorithm for POP (removing an item from the stack) operation

This algorithm deletes the top element of the stack and assigns it to a variable element

1. Check for the stack Underflow as

   If top<0 then

   Print "Stack Underflow" and Exit the program

   else

   Remove the top element and set this element to the variable as

   Set element=Stack [top]

   Decrement top by 1 as

   Set top=top-1
2. Print 'element' as a deleted item from the stack
3. Stop

C function for PUSH operation

We can define the push function as give below:

```
void push( )
{
        int item;
        if(top == MAXSIZE - 1)  //Checking stack overflow
                printf("\n The Stack Is Full");
        else
        {
                printf("Enter the element to be inserted");
                scanf("%d", &item);         //reading an item
                top= top+1;                 //increase top by 1
                stack[top] = item; //storing the item at the top of the stack
        }
}
```

The C function for POP operation

Alternatively, we can define the POP function as give below:

```
void pop( )
{
        int item;
        if(top < 0)  //Checking Stack Underflow
```

```
        printf("The stack is Empty");
    else
    {
        item = stack[top];  //Storing top element to item variable
        top = top-1;                //Decrease top by 1
        printf("The popped item is=%d", item); //Displaying the deleted items
    }
}
```

Complete menu driven program in C to array implementation of stack

```
#include<stdio.h>
#include<conio.h>
#define MAX 100
struct stack
{
    int item[MAX];
    int tos;
};
typedef struct stack st;
void push(st*, int);
int pop(st*);
void display(st*);
void main( )
{
    int data, ch, x;
    st *s;
    clrscr( );
    s→tos=-1;        // make empty stack
    printf("---------MENU FOR PROGRAM----------\n");
    printf("1:push\n2:pop\n3:display\n4:exit\n");
    do
    {
        printf("Enter your choice\n");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1:
                printf("Enter data to be inserted\n");
                scanf("%d", &data);
                push(s, data);
                break;
            case 2:
                x=pop(s);
```

```
                        printf("\n poped item is:");
                        printf("%d\n", x);
                        break;
                case 3:
                        display(s);
                        break;
                default:
                        exit(1);
            }
        }while(ch<4);
        getch();
}
void push(st *s, int d) /*PUSH function*/
{
        if(s→tos==MAX-1)
        {
                printf("Stack is full\n");
        }
        else
        {
                ++s→tos;
                s→item[s→tos]=d;
        }
}
int pop(st *s) /*POP function*/
{
        int itm;
        if(s->tos==-1)
        {
                printf("Stack is empty\n");
                return(0);
        }
        else
        {
                itm=s→item[s→tos];
                s→tos--;
                return(itm);
        }
}
void display(st *s) /*display function*/
{
        int i;
        if(s→tos==-1)
```

```
        printf("There is no data item to display\n");
    else
    {
        for(i=s→tos; i>=0; i--)
        {
            printf("%d\t", s→item[i]);
        }
    }
}
```

Output

```
1: Push
2: Pop
3: Display
Enter your choice
1
Enter data item to be inserted
99
Enter your choice
1
Enter data item to be inserted
89
Enter your choice
1
Enter data item to be inserted
88
Enter your choice
1
Enter data item to be inserted
90
Enter your choice
3
Stack elements are:
99 89 88 90 Enter your choice
2
Popped element=90
Enter your choice
2
Popped element=88
Enter your choice
2
Popped element=89
Enter your choice
1
```

Enter data item to be inserted
45
Enter your choice
3
Stack elements are:
99 45 Enter your choice

Application of stack: Delimiter Matching

One application of the stack is in matching delimiters in a program. This is an important example because delimiter matching is part of any compiler: No program is considered correct if the delimiters are mismatched. In C programs, we have the following delimiters: parentheses "(" and ")", square brackets "[" and "]", curly brackets "{" and "}", and comment delimiters "/*" and "*/".

Example: Delimiter Matching

The delimiters are the braces '{'and'}', brackets '['and']', and parentheses '('and')'. Each opening or left delimiter should be matched by a closing or right delimiter; that is, every '{' should be followed by a matching '}' and so on. Also, opening delimiters that occur later in the string should be closed before those occurring earlier.

Delimiter matching algorithm

```
1. Start
2. read one character at a time say 'ch' from input file
3. while not end of file
      a.  if ch is '(' , '[', or '{'
                Push (ch);
      b.  else if ch is '/'
                read the next character;
                if this character is '*'
                      skip all characters until "*/" is found and report an error
                      if the end of file is reached before "*/" is encountered;
                else ch = the character read in;
                      Continue;        // go to the beginning of the loop;
      c.  else if 'ch' is ')', ']', or '}'
                if 'ch' and popped off delimiter do not match failure;
      d.  read next character 'ch' from file; // else ignore other characters;
4. if stack is empty
            Display message "success";
5. else
            Display message "failure";
6. Stop
```

Tracing: Processing the statement s=a[5]+b/(c*(d+e))+(f/g); with the algorithm delimiter matching.

Stack	Nonblank Character Read	Input Left
Φ	.....	s=a[5]+b/(c*(d+e))+(f/g);
Φ	s	=a[5]+b/(c*(d+e))+(f/g);
Φ	=	a[5]+b/(c*(d+e))+(f/g);
Φ	a	[5]+b/(c*(d+e))+(f/g);
[	[	5]+b/(c*(d+e))+(f/g);
[	5	]+b/(c*(d+e))+(f/g);
Φ	]	+b/(c*(d+e))+(f/g);
Φ	+	b/(c*(d+e))+(f/g);
Φ	b	/(c*(d+e))+(f/g);
Φ	/	(c*(d+e))+(f/g);
(	(	c*(d+e))+(f/g);
(	c	*(d+e))+(f/g);
(	*	(d+e))+(f/g);
((	(	d+e))+(f/g);
((	d	+e))+(f/g);
((	+	e))+(f/g);
((	e	))+(f/g);
(	)	)+(f/g);
Φ	)	+(f/g);
Φ	+	(f/g);
(	(	f/g);
(	f	/g);
(	/	g);
(	g	);
Φ	)	;
Φ	;	Φ

Linked list implementation of stack (or dynamic implementation of stack)

Stack can be implemented using both arrays and linked lists. The limitation, in the case of an array, is that we need to define the size at the beginning of the implementation. This makes our stack static. It can also result in "stack overflow" if we try to add elements after the array is full. So, to alleviate this problem, we use a linked list to implement the stack so that it can grow in real time.

Structure for linked list implementation of stack

```
Struct Linked_stack
{
        int info;
```

```
        Linked_stack *next;
}
typedef struct Linked_stack NodeType;
Linked_stack  *top;
top=NULL;
```

PUSH operation

We can use the following steps to insert a new node into the stack:

1. Create a newNode with given value.
2. Check whether stack is Empty (top == NULL)
3. If it is Empty, then set newNode → next = NULL.
4. If it is Not Empty, then set newNode → next = top.
5. Finally, set top = newNode.

The time complexity for the **Push** operation is O(1). The method for push will look like the following:
Let, *top be the top of the stack or pointer to the first node of the list.

```
void push(item)
{
        NodeType  *nnode;
        int data;
        nnode=(NodeType *)malloc(sizeof(NodeType));
        if(top==0)
        {
                nnode→info=item;
                nnode→next=NULL;
                top=nnode;
        }
        else
        {
                nnode→info=item;
                nnode→next=top;
                top=nnode;
        }
}
```

POP Operation

To pop an item from a linked stack, we just have to reverse the operation. Let *top be the top of the stack or pointer to the first node of the list. We can use the following steps to delete a node from the stack:

1. Check whether stack is Empty (top == NULL).
2. If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
3. If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
4. Then set 'top = top → next'.
5. Finally, delete 'temp' (free (temp)).

The time complexity for Pop operation is O(1). The method for pop will look like the following:

```
void pop( )
{
        NodeType *temp;
        if(top==0)
        {
                printf("Stack contain no elements:\n");
                return;
        }
        else
        {
                temp=top;
                top=top→next;
                printf("\n deleted item is %d\t", temp→info);
                free(temp);
        }
}
```

Display: Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack:

1. Check whether stack is Empty (top == NULL).
2. If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
3. If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
4. Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack (temp → next! = NULL).
5. Finally, Display 'temp → data ---> NULL'

A Complete menu driven program in C for linked list implementation of stack

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>
struct node
{
        int info;
        struct node *next;
};
typedef struct node NodeType;
NodeType *top;
top=0;
void push(int);
void pop();
void display();
void main( )
```

```
{
        int choice, item;
        clrscr( );
        do
        {
                printf("\n1.Push \n2.Pop \n3.Display\n4:Exit\n");
                printf("enter ur choice\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:
                                printf("\nEnter the data:\n");
                                scanf("%d",&item);
                                push(item);
                                break;
                        case 2:
                                pop();
                                break;
                        case 3:
                                display();
                                break;
                        case 4:
                                exit(1);
                                break;
                        default:
                                printf("invalid choice\n");
                }
        }while(choice<5);
        getch();
}

void push(int item)  /*PUSH function*/
{
        NodeType  *nnode;
        int data;
        nnode=( NodeType *)malloc(sizeof(NodeType));
        if(top==0)
        {
                nnode->info=item;
                nnode->next=NULL;
                top=nnode;
        }
        else
```

```
        {
                nnode->info=item;
                nnode->next=top;
                top=nnode;
        }
}
void pop( ) /*POP function*/
{
        NodeType *temp;
        if(top==0)
        {
                printf("Stack contain no elements:\n");
                return;
        }
        else
        {
                temp=top;
                top=top->next;
                printf("\ndeleted item is %d\t",temp->info);
                free(temp);
        }
}
void display() /*display function*/
{
        NodeType *temp;
        if(top==0)
        {
                printf("Stack is empty\n");
                return;
        }
        else
        {
                temp=top;
                printf("Stack items are:\n");
                while(temp!=0)
                {
                        printf("%d\t",temp->info);
                        temp = temp->next;
                }
        }
}
```

Output

```
1: Push
2: Pop
3: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
44
Enter your choice
1
Enter data item to be inserted
55
Enter your choice
3
55 44 22 Enter your choice
2
Popped item=55
Enter your choice
2
Popped item=44
Enter your choice
2
Popped item=22
Enter your choice
2
Stack empty
```

Infix, Prefix and Postfix Notation

One of the applications of the stack is to evaluate the expression. We can represent the expression following three types of notation:

- Infix expression
- Prefix expression
- Postfix expression

Infix expression

It is an ordinary mathematical notation of expression where operator is written in between the operands. Example: A+B. Here **+** is an **operator** and '*A*' and '*B*' are called **operands.**

Prefix notation

In prefix notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notation.

Example: +AB

Postfix notation

In postfix notation the operators are written after the operands so it is called the postfix notation (post mean after). In this notation the operator follows the two operands.

Example: AB+

Note: Both prefix and postfix is parenthesis free expressions. For example

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Precedence rule

While converting infix to postfix you have to consider the **precedence rule**, and the precedence rules are as follows. To study further, we must also know the operator precedence and their associatively:

Token	Operator	Precedence	Associativity
() [] -.	function call array element struct or union member	17	left-to-right
– ++	increment, decrement	16	left-to-right
! ~ - + & * sizeof	logical NOT one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise AND	8	left-to-right
^	bitwise XOR	7	left-to-right
|	bitwise OR	6	left-to-right
&&	logical AND	5	left-to-right
||	logical OR	4	left-to-right

?:	conditional	3	right-to-left
= += /= *= %= &= ^=	assignment	2	right-to-left
,	comma	1	left-to-right

Exponentiation (the expression A$B is A raised to the B power, so that 3$2=9). When unparenthesized operators of the same precedence are scanned, the order is assumed to be left to right except in the case of exponentiation, where the order is assumed to be from right to left.

- A+B+C means (A+B)+C
- ABC means A$(B$C)

Consider an example that illustrate the converting of infix to postfix expression, A + (B*C). Use the following **rule** to convert it in postfix:

- Parenthesis for emphasis
- Convert the multiplication
- Convert the addition
- Postfix form

Illustration

A + (B * C) Infix form

A + (B * C) Parenthesis for emphasis

A + (BC*) Convert the multiplication

A (BC*) + Convert the addition

ABC*+ Post-fix form

Consider an example

(A + B) * ((C - D) + E) / F Infix form

= (AB+) * ((C – D) + E) / F

= (AB+) * ((CD-) + E) / F

= (AB+) * (CD-E+) / F

= (AB+CD-E+*) / F =AB+CD-E+*F/ Postfix form

Algorithm to convert infix to postfix notation

Let here two stacks opstack and poststack are used and otos & ptos represents the opstack top and poststack top respectively. The opstack is used to store operators of given expression and poststack is used to store converted corresponding postfix expression.

1. Start
2. Scan one character at a time of an infix expression from left to right
3. Opstack=the empty stack
4. Repeat till there is data in infix expression

   4.1 If scanned character is '('then push it to opstack

   4.2 If scanned character is operand then push it to poststack

   4.3 If scanned character is operator then

   if (opstack!=-1)

   While (precedence (opstack [otos])>precedence (scan character)) then

pop and push it into poststack

Otherwise

Push scanned character into opstack

4.4 If scanned character is ')' then

Pop and push into poststack until '('is not found and ignore both symbols

5. Pop and push into poststack until opstack is not empty.

6. Stop

Trace of Conversion Algorithm

The following tracing of the algorithm illustrates convert infix to postfix the algorithm. Consider an infix expression

((A-(B+C))*D)$(E+F)

Scan character	Poststack	opstack
(	.....	(
(	.....	((
A	A	((
-	A	((-
(	A	((-(
B	AB	((-(
+	AB	((-(+
C	ABC	((-(+
)	ABC+	((-
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	.....
$	ABC+-D*	$
(	ABC+-D*	$(
E	ABC+-D*E	$(
+	ABC+-D*E	$(+
F	ABC+-D*EF	$(+
)	ABC+-D*EF+	$
.....	ABC+-D*EF+$ (postfix)	.....

Converting an Infix expression to prefix expression

The precedence rule for converting an expression from infix to prefix is identical to that of infix to postfix. Only changes from postfix conversion are that the operator is placed before the operands rather than after them. The prefix of A+B-C is -+ABC.

Example: Consider an infix expression:

A $ B * C – D + E / F / (G + H) infix form

= A $ B * C – D + E / F / (+GH)
= $AB* C – D + E / F / (+GH)
= *$ABC-D+E/F/ (+GH)
= *$ABC-D+ (/EF)/ (+GH)
= *$ABC-D+//EF+GH
= (-*$ABCD) + (//EF+GH)
= +-*$ABCD//EF+GH which is in prefix form.

Complete program in C to convert an expression from infix to postfix

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
int precedency(char);
void main()
{
        int i, otos=-1, ptos=-1, len, length;
        char infix[100], poststack[100], opstack[100];
        printf("Enter a valid infix\n");
        gets (infix);
        length=strlen(infix);
        len=length;
        for(i=0;i<=length-1;i++)
        {
                if(infix[i]=='(')
                {
                        opstack[++otos]=infix[i];
                        len++;
                }
                else if(isalpha(infix[i]))
                {
                        poststack[++ptos]=infix[i];
                }
                else if (infix[i]==')')
                {
                        len++;
                        while(opstack[otos]!='(')
                        {
                                poststack[++ptos]=opstack[otos];
                                otos--;
                        }
                        otos--;
                }
                else  //operators
                {
                        if(precedency(opstack[otos])>precedency(infix[i]))
                        {
                                poststack[++ptos]=opstack[otos--];
                                opstack[++otos]=infix[i];
```

```
                }
                opstack[++otos]=infix[i];
            }
        }
        while(otos!=-1)
        {
            poststack[++ptos]=opstack[otos];
            otos--;
        }
        for(i=0; i<len; i++)   /*for displaying*/
        {
            printf("%c", poststack[i]);
        }
        getch();
}
int precedency(char ch)  /*precedence function*/
{
        switch(ch)
        {
            case '$':
                return(4);
            case'*':
            case'/':
                return(3);
            case'+':
            case'-':
                return(2);
            default:
                return(1);
        }
}
```

Output

Enter valid infix expression
(A+B*C/D)*(E-F/G)
ABCD/*+EFG/-*

Evaluating the Postfix expression

Each operator in a postfix expression refers to the previous two operands in the expression. To evaluate the postfix expression, we use the following procedure:

- Each time we read an operand we push it onto a stack.
- When we reach an operator, its operands will be the top two elements on the stack.
- We can then pop these two elements perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.

Consider an example

3 4 5 * +
=3 20 + =23 (answer)

Evaluating the given postfix expression

6 2 3 + - 3 8 2 / + * 2 $ 3 +
=6 5 - 3 8 2 / + * 2 $ 3 +
=1 3 8 2 / + * 2 $ 3 +
=1 3 4 + * 2 $ 3 +
=1 7 * 2 $ 3 +
=7 2 $ 3 +
=49 3 +
= 52

Algorithm to evaluate the postfix expression

Here we use only one stack called vstack (value stack).

1. Scan one character at a time from left to right of given postfix expression
   1.1 if scanned symbol is operand then
      - read its corresponding value and push it into vstack
   1.2 if scanned symbol is operator then
      - pop and place into op2
      - pop and place into op1
      - Compute result according to given operator and push result into vstack
2. Pop and display which is required value of the given postfix expression
3. Stop

Trace of Evaluation

Consider an example to evaluate the postfix expression tracing the algorithm

ABC+*CBA- +*

123+*321-+*

Scanned character	Value	Op2	Op1	Result	vstack
A	1	.....	.....	.....	1
B	2	.....	.....	.....	1 2
C	3	.....	.....	.....	1 2 3
+	.....	3	2	5	1 5
*	.....	5	1	5	5
C	3	.....	.....	.....	5 3
B	2	.....	.....	.....	5 3 2
A	1	.....	.....	.....	5 3 2 1
-	.....	1	2	1	5 3 1
+	.....	1	3	4	5 4
*	.....	4	5		20

Its final value is 20.

Evaluating the Prefix Expression

To evaluate the prefix expression, we use two stacks and some time it is called two stacks algorithms. One stack is used to store operators and another is used to store the operands. Consider an example

+ 5 *3 2 prefix expression

= +5 6

=11

Evaluate the given prefix expression

/+ 5 3 - 4 2 prefix equivalent to (5+3)/ (4-2) infix notation

= / 8 - 4 2

= / 8 2 = 4

Complete program in C for evaluating postfix expression

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
void push(int);
int pop();
int vstack[100];
int tos=-1;
void main()
{
        int i, res, len, op1, op2, value[100];
        char postfix[100], ch;
        clrscr( );
        printf("Enter a valid postfix\n");
        gets(postfix);
        len=strlen(postfix);
        for(i=0;i<=len-1;i++)
        {
                if(isalpha(postfix[i]))
                {
                        printf("Enter value of %c",postfix[i]);
                        scanf("%d",&value[i]);
                        push(value[i]);
                }
                else
                {
                        ch=postfix[i];
                        op2=pop();
                        op1=pop();
                        switch(ch)
                        {
                                case '+':
                                        push(op1+op2);
                                        break;
```

```
                    case'-':
                            push(op1-op2);
                            break;
                    case'*':
                            push(op1*op2);
                            break;
                    case'/':
                            push(op1/op2);
                            break;
                    case'$':
                            push(pow(op1,op2));
                            break;
                    case'%':
                            push(op1%op2);
                            break;
                }
            }
        }
        printf("The result is:");
        res=pop( );
        printf("%d", res);
        getch( );
}
void push(int val)  /*insertion function*/
{
        vstack[++tos]=val;
}
int pop( )  /*deletion function*/
{
        int n;
        n=vstack[tos--];
        return(n);
}
```

Output

```
Enter valid postfix expression
ab*c-
Enter value of a
10
Enter value of b
2
Enter value of c
5
The result is: 15
```

Exercise

1. What is stack? How it is differ from queue? Show that stack and queue as ADT.
2. How you can convert given infix expression into their equivalent postfix expression?
3. Evaluate following postfix expressions by using stack:
   a. A B - C D E F / + G - * - H -
   b. A B C D * - E / + F + G H / -
   c. A B + C * D E F G * + / + H -

   Where, A=4, B=8, C=2, D=5, E=6, F=9, G=1, H=3
4. Convert following infix expressions into prefix and postfix expressions:
   (A * (B + (C / D)))
5. Write a program to add any number of large integers. The problem can be approached in at least two ways.
   a. First, add two numbers and then repeatedly add the next number with the result of the previous addition.
   b. Create a vector of stacks and then use a generalized version of addingLargeNumbers() to all stackes at the same time.
6. Write a program that determines whether an input string is a palindrome; that is, whether it can be read the same way forward and backward. At each point, you can read only one character of the input string; do not use an array to first store this string and then analyze it (except, possibly, in a stack implementation).Consider using multiple stacks.
7. Write down basic operations of stack. And also describe push and pop function of stack.
8. What is main concept behind recursion? Write a program in C to find reverse of given string by using recursion.
9. Put the elements on the stack S in ascending order using one additional stack and some additional non-array variables.
10. What are the basic operations of stack? Describe them with suitable C program.
11. Show that stack as an ADT.
12. Trace the algorithm to convert given infix to postfix expressions:
    a. (A + B) * C + D / (E + F * G) – H
    b. A + ((B - C * D) / E) + F - G / H
    c. (A * B + C) / D - E / (F + G)
    d. ((H * ((((A + ((B + C) * D)) * F) * G) * E)) + J)
13. What are the application areas of stack in computer application fields?
14. What is dynamic programming? How it is differ from greedy algorithm?
15. What is Stack and where it can be used?
16. What are Infix, prefix, Postfix notations?
17. What are the various operations that can be performed on different Data Structures?
18. Write down an algorithm to convert given infix to postfix expression.
19. Write down an algorithm to evaluate given postfix expression.
20. Transfer elements from stack S_1 to stack S_2 so that the elements from S_2 are in the same order as on S_1
    a. Using one additional stack
    b. Using no additional stack but only some additional non-array variables

•••

Chapter

5

Recursion

Introduction

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result. In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition.

For the problems solve recursively following two conditions must be satisfied:

- Each time a function calls itself and it must be closer to a solution.
- The problem statement must include a stopping condition.

Example: Finding factorial of a given number

Let us consider the factorial of a number and its algorithm described recursively:

We know that n! = n * (n-1)!

(n-1)! = n-1 * (n-2)! And so on up to 1.

Algorithm

```
Factorial (n)
{
        if n==1
```

```
                return 1
        else
                return n * Factorial (n-1)
}
```

Let's trace the evaluation of factorial (5)

Factorial (5) =
5*Factorial (4) =
5*(4*Factorial (3)) =
5*(4*(3*Factorial (2))) =
5*(4*(3*(2*Factorial (1)))) =
5*(4*(3*(2*(1*Factorial (0))))) =
5*(4*(3*(2*(1*1)))) =
5*(4*(3*(2*1))) =
5*(4*(3*2)) =
5*(4*6) =
5*24 =
120

Divide-and-conquer algorithms

An important problem-solving technique that makes use of recursion is dividing and conquers. A **divide-and-conquer algorithm** is an efficient recursive algorithm that consists of two parts:

- Divide, in which smaller problems are solved recursively (except, of course, base cases)
- Conquer, in which the solution to the original problem is then formed from the solutions to the sub problems

Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer algorithms, whereas routines whose text contains only one recursive call are not. Consequently, the recursive routines presented so far in this chapter are not divide-and-conquer algorithms. Also, the sub problems usually must be disjoint (i.e., essentially none overlapping), so as to avoid the excessive costs seen in the sample recursive computation of the Fibonacci numbers.

Example 1: Calculation of the factorial of an integer number using recursive function

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n;
        long int facto;
        long int factorial(int n);
        printf("Enter value of n:");
        scanf("%d", &n);
        facto=factorial(n);
        printf("%d! = %ld", n, facto);
        getch( );
}
```

```
long int factorial(int n)
{
        if(n == 0)
                return 1;
        else
                return n * factorial(n-1);
}
```

Output

Enter value of n:

6

Factorial of 6=720

Example 2: Program to find factorial of an integer number without using recursive function

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int n;
        long int facto;
        long int factorial(int n);
        printf("Enter value of n:");
        scanf("%d", &n);
        facto=factorial(n);
        printf("%d! = %ld", n, facto);
        getch();
}
long int factorial(int n)
{
        long int facto=1;
        int i;
        if(n==0)
                return 1;
        else
        {
                for(i=1;i<=n;i++)
                        facto=facto*i;
                return facto;
        }
}
```

Output

Enter value of n:

6

Factorial of 6=720

Example 3: Program to generate Fibonacci series up to n terms using recursive function. (Hint Fibonacci sequence=0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...)

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int n,i;
        int fibo(int);
        printf("Enter n:");
        scanf("%d", &n);
        printf("Fibonacci numbers up to %d terms:\n", n);
        for(i=1;i<=n;i++)
                printf("%d\n", fibo(i));
        getch( );
}
int fibo(int k)
{
        if(k == 1 || k == 2)
                return 1;
        else
                return fibo(k-1)+fibo(k-2);
}
```

Output

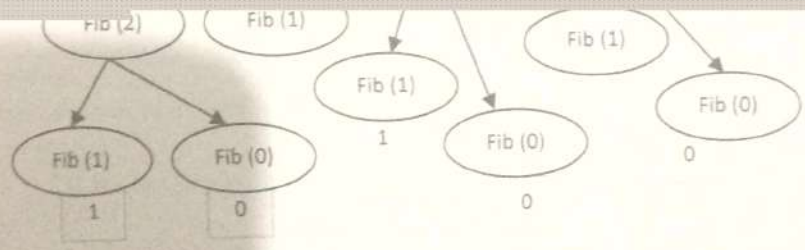
```
Enter value of n:
7
Fibonacci numbers up7th term is:
1 1 2 3 5 8 13
```

Example 4: Program to find nth tern of Fibonacci series using recursion

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int n,i;
        int fibo(int);
        printf("Enter value of n:");
        scanf("%d", &n);
        printf("nth Fibonacci term is:\n");
        printf("%d", fibo(n));
        getch( );
}
int fibo(int k)
{
```

on. (Hint

```
if(k == 1 || k == 2)
        return 1;
else
        return fibo(k-1)+fibo(k-2);
```



Fifth term of Fibonacci series=1+0+1+1+0+1+0+1=5

Example 5: Program to find sum of first n natural numbers using recursion

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

```
{
        if(n == 1)
                return 1;
        else
                return n + sum_natural(n-1);
}
```

Example 6: Recursive java program to display following pattern

```
*
*       *
*       *       *
*       *       *       *
*       *       *       *       *
*       *       *       *       *       *
*       *       *       *       *       *       *
```

```
#include<stdio.h>
#include<conio.h>
void Triangle(int n)
{
        if (n <= 0)
                return;
        else
                Triangle(n - 1);
        for (int i = 1; i <= n; i++)
        {
                printf("*\t");
        }
        printf("\n");
}
int main( )
{
        Triangle(7);
        return 0;
}
```

The trace of the program will be as follows:

```
Triangle (7)
 Triangle (6)
  Triangle (5)
   Triangle (4)
    Triangle (3)
     Triangle (2)
      Triangle (1)
```

Triangle (0) ← base case
Triangle (1) ← prints 1 star & new line
Triangle (2) ← prints 2 stars & new line
Triangle (3) ← prints 3 stars & new line
Triangle (4) ← prints 4 stars & new line
Triangle (5) ← prints 5 stars & new line
Triangle (6) ← prints 6 stars & new line
Triangle (7) ← prints 7 stars & new line

Tower of Hanoi problem

Initial state

- There are three poles named as origin, intermediate and destination.
- **n** number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.
- The disks are numbered as 1, 2, 3, 4… n.

Objective

- Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

Conditions

- Move only one disk at a time.
- Each disk must always be placed around one of the pole.
- Never place larger disk on top of smaller disk.

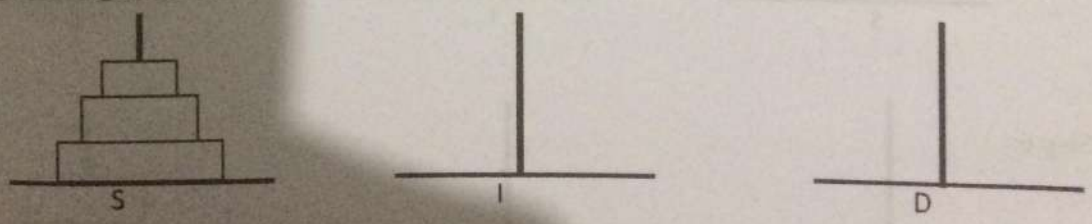
Algorithm

To move a tower of *n* disks from **source** to **destination** (where *n* is positive integer)

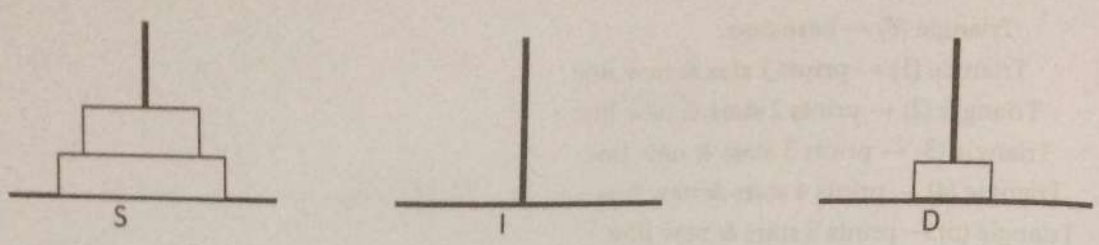
1. If *n*==1:
   1.1. Move a single disk from **source** to **destination**
2. If n > 1:
   2.1. Let **temp** be the remaining pole other than **source** and **destination**
   2.2. Move a tower of (*n*– 1) disks form **source** to **temp**
   2.3. Move a single disk from **source** to **destination**
   2.4. Move a tower of (*n*– 1) disks form **temp** to **destination**
3. Stop

Tracing (Tracing for n=3)

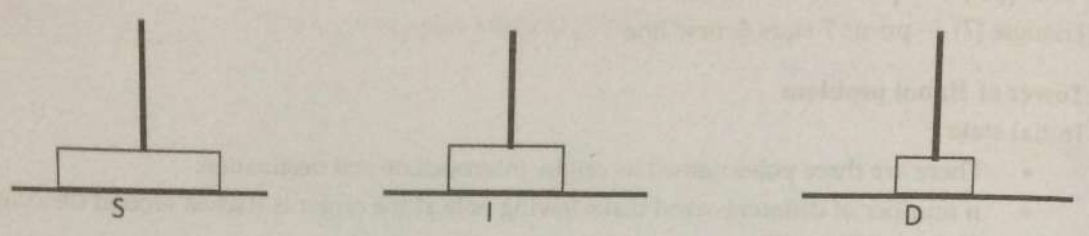
Step 0:



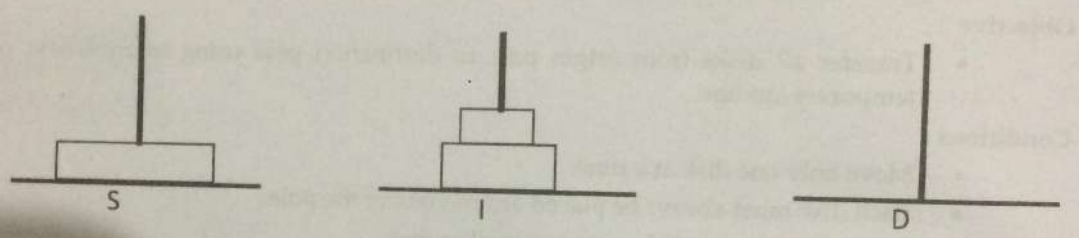
Step 1:



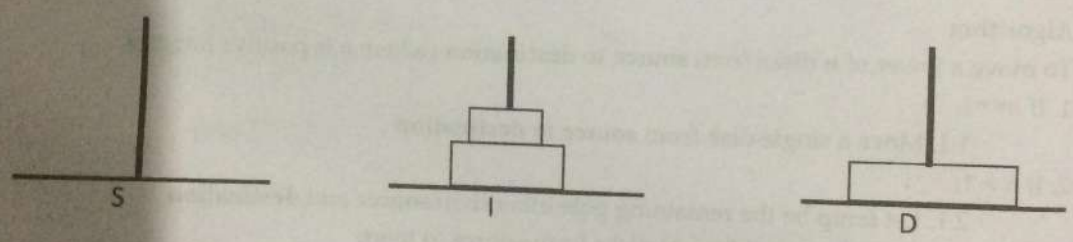
Step 2:



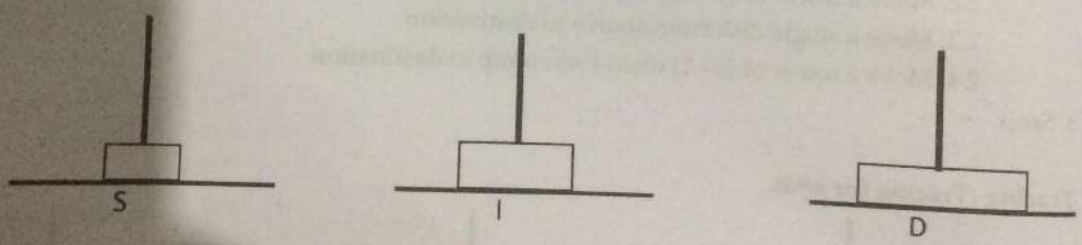
Step 3:



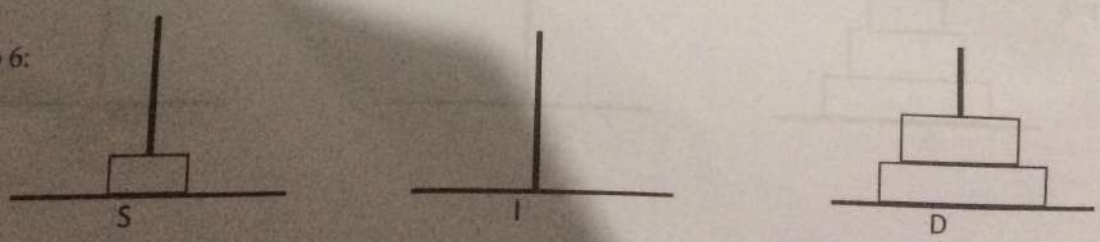
Step 4:



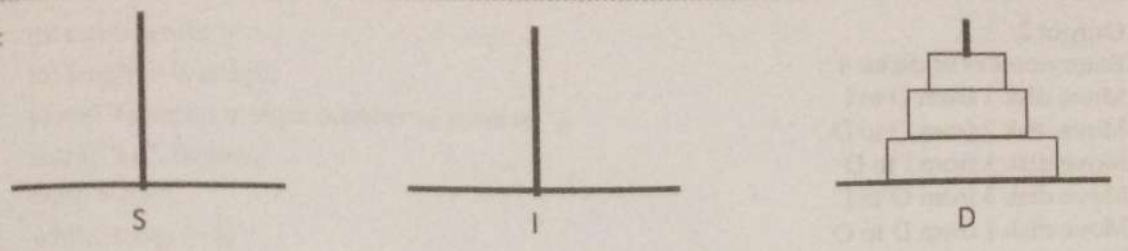
Step 5:



Step 6:



Step 7:



Number of steps required in TOH problem= $(2^n -1)$

For n=3,

Number of steps required in TOH problem= $(2^n -1) = (2^3 -1) = 8-1=7$

Example 7: Recursive solution of tower of Hanoi

```
#include <stdio.h>
#include <conio.h>
void TOH(int, char, char, char); //Function prototype
void main( )
{
        int n;
        printf("Enter number of disks");
        scanf("%d",&n);
        TOH(n,'O','D','I');
        getch( );
}
void TOH(int n, char A, char B, char C)
{
        if(n>0)
        {
                TOH(n-1, A, C, B);
                printf("Move disk %d from %c to %c\n", n, A, B);
                TOH(n-1, C, B, A);
        }
}
```

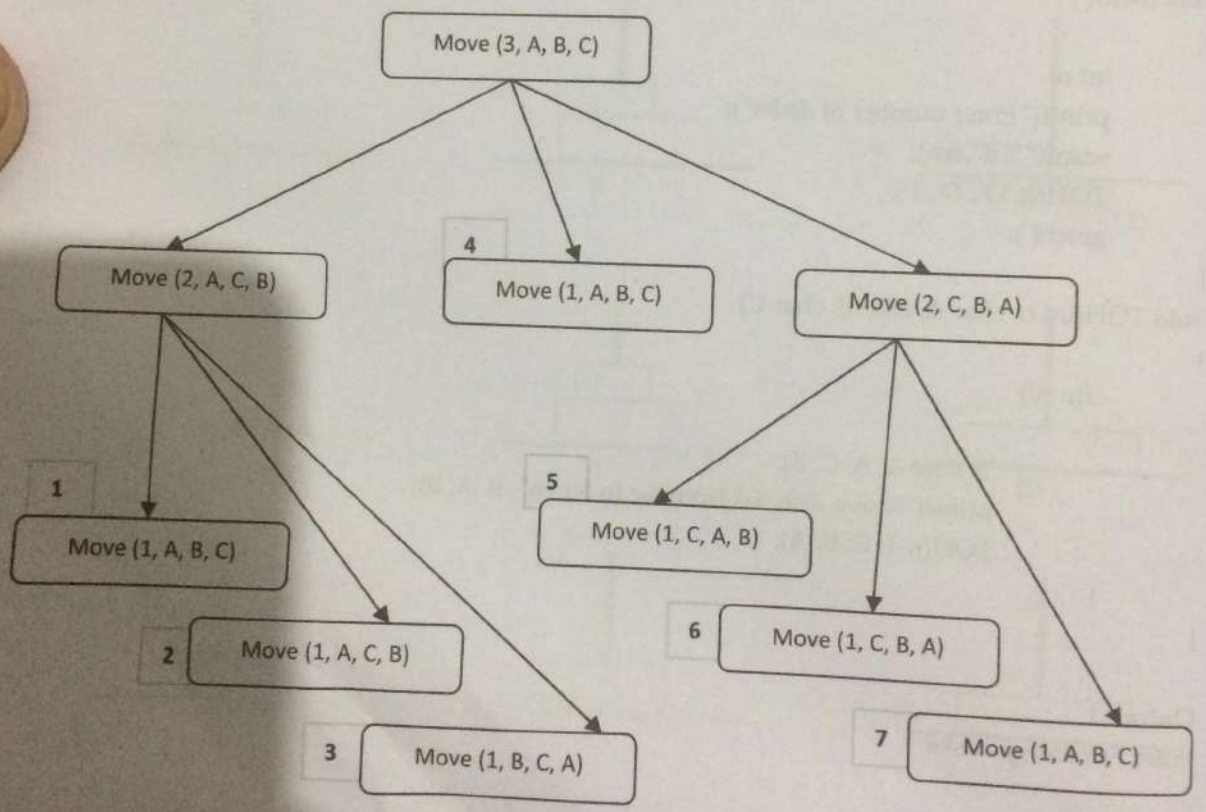
Output 1

```
Enter number of disks
3
Move disk 1 from O to D
Move disk 2 from O to I
Move disk 1 from D to I
Move disk 3 from O to D
Move disk 1 from I to O
Move disk 2 from I to D
Move disk 1 from O to D
```

Output 2

Enter number of disks: 4
Move disk 1 from O to I
Move disk 2 from O to D
Move disk 1 from I to D
Move disk 3 from O to I
Move disk 1 from D to O
Move disk 2 from D to I
Move disk 1 from O to I
Move disk 4 from O to D
Move disk 1 from I to D
Move disk 2 from I to O
Move disk 1 from D to O
Move disk 3 from I to D
Move disk 1 from O to I
Move disk 2 from O to D
Move disk 1 from I to D

Program Tracing



Example 8: Program to find the reverse of given number by using recursion

```
#include <stdio.h>
#include <math.h>
int rev(int, int);
int main()
{
```

```
    int num, result;
    int length = 0, temp;
    printf("Enter an integer number to reverse: ");
    scanf("%d", &num);
    temp = num;
    while (temp != 0)
    {
        length++;
        temp = temp / 10;
    }
    result = rev(num, length);
    printf("The reverse of %d is %d.\n", num, result);
    return 0;
}
int rev(int num, int len)
{
    if (len == 1)
    {
        return num;
    }
    else
    {
        return (((num % 10) * pow(10, len - 1)) + rev(num/10, --len));
    }
}
```

Example 9: Program to find the reverse of given string by using recursion

```
#include <stdio.h>
void Reverse( );
int main( )
{
    printf("Enter a sentence: ");
    Reverse( );
    return 0;
}
void Reverse( )
{
    char ch;
    scanf("%c", &ch);
    if( ch!= '\n')
        Reverse( );
    printf("%c", ch);
}
```

Example 10: Program to find the Highest common factor (greatest common divisor) of any two numbers by using recursion

```
#include <stdio.h>
int GCD(int, int );
int main( )
{
        int a, b, g;
        printf("Enter any two numbers");
        scanf("%d%d", &a, &b);
      g=GCD(a, b);
      printf("GCD of given two numbers=%d", g);
        return 0;
}
int GCD(int x, int y)
{
        if(y==0)
                return x;
        else
                return (GCD(y, x%y));
}
```

Example 11: Program to find the value of x^n, (where x and n are any two numbers) by using recursion.

```
#include <stdio.h>
int power(int, int );
int main( )
{
        int x, n, p;
        printf("Enter value of x and n");
        scanf("%d%d", &x, &n);
      p=power(x, n);
      printf("The result of x^n is%d", p);
        return 0;
}
int power(int x, int n)
{
        if(n==0)
                return 1;
        else
                return (x*power(x, n-1));
}
```

Example 12: Program to display string "MY name is Khan" 10 times by using recursion.

```
#include <stdio.h>
int main( )
{
        static int i=10;
        if(i>0)
        {
                printf("My name is Khan\n");
```

```
            i--;
            main( );
        }
}
```

Output

```
My name is Khan
My name is Khan
My name is Khan
My name is Khan
My name is Khan
My name is Khan
My name is Khan
My name is Khan
My name is Khan
My name is Khan
```

Example 13: Program to check whether given string is palindrome or not by using recursion. (Note: string "madam" is palindrome but string "Bhupi" is not palindrome)

```
#include <stdio.h>
#include <string.h>
void checkpalendrome(char [ ], int);
int main( )
{
        char word[15];
        printf("Enter a word to check if it is a palindrome\n");
        scanf("%s", word);
        check(word, 0);
        return 0;
}
void checkpalendrome(char word[ ], int index)
{
        int len;
        len = strlen(word) - (index + 1);
        if (word[index] == word[len])
        {
                if (index + 1 == len || index == len)
                {
                        printf("The entered word is a palindrome\n");
                        return;
                }
                checkpalendrome (word, index + 1);
        }
        else
                printf("The entered word is not a palindrome\n");
}
```

Example 14: Program to check whether given number is palindrome or not by using recursion. (**Note**: number 121 is palindrome but 5678 is not palindrome)

```
#include<stdio.h>
int main( )
{
        int num,rev;
        printf("\n Enter a number :");
        scanf("%d", &num);
        rev=reverse(num);
        printf("\n After reverse the no is :%d", rev);
        return 0;
}
int reverse(int num)
{
        static int r, sum=0;
        if(num!=0)
        {
                r=num%10;
                sum=sum*10+r;
                reverse(num/10);
        }
        else
                return sum;
}
```

Advantages of Recursion

- The code may be much easier to write.
- To solve some problems which are naturally recursive such as tower of Hanoi.

Disadvantages of Recursion

- Recursive functions are generally slower than non-recursive functions.
- May require a lot of memory to hold intermediate results on the system stack.
- It is difficult to think recursively so one must be very careful when writing recursive functions.

Types of recursion

Recursion can be categorized into following 5 types:

- Direct recursion
- Indirect recursion
- Tail recursion
- Linear recursion
- Tree recursion

Direct recursion

A function is called direct recursive if it calls directly to itself until some base condition is not satisfied.

Example: Function for finding factorial of given number by using direct recursion method.

```
int fact(int num)
{
        if(num==0)
                return 1;
        else
                return num*fact(num-1);
}
```

Indirect recursion

Up to now we discussed only direct recursion, where a method f() called itself. However, f() can call itself indirectly via a chain of other calls. For example, f() can call g(), and g() can call f(). This is the simplest case of indirect recursion. The chain of intermediate calls can be of an arbitrary length, as in:

$$f() \rightarrow f1() \rightarrow f2() \rightarrow \cdots \rightarrow fn() \rightarrow f()$$

Simply, a function is said to be indirect recursive if it calls to another function and again this new function calls to its calling function.

Example: Program for Indirect recursion in C

```
int fun1(int x)
{
        if(x<=0)
                return 1;
        else
                return fun2(x);
}
int fun2(int y)
{
        return fun1(y-1);
}
```

Tail recursion

A function that returns the value of its recursive call is said to be **tail recursive.** Simply a function is said to be tail recursive if there are no any calculations occur in the recursive stage and it only returns the value.

Example: Complete program in C to showing the use of tail recursion

```
Fact (n, accumulator)
{
        if (n == 0)
                return accumulator;
        else
                return Fact(n - 1, n * accumulator);
}
Factorial (n)
```

```
{
        return Fact(n, 1);
}
void main( )
{
        int num, f;
        printf("Enter any number");
        scanf("%d", &num);
        f=Factorial (num);
        printf ("Factorial of given number is %d", f);
        getch( );
}
```

Output

Enter any number

10

Factorial of given number= 3628800

Backtracking

Backtracking is a form of recursion. But it involves choosing only option out of any possibilities. We begin by choosing an option and backtrack from it, if we reach a state where we conclude that this specific option does not give the required solution. We repeat these steps by going across each available option until we get the desired solution.

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called **backtracking,** and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

Let's take a standard problem.

N-Queens Problem: Given a chess board having n×n cells, we need to place n queens in such a way that no queen is attacked by any other queen. A queen can attack horizontally, vertically and diagonally.

So initially we are having n×n un-attacked cells where we need to place n queens. Let's place the first queen at a cell (i, j), so now the number of un-attacked cells is reduced, and number of queens to be placed is n−1. Place the next queen at some un-attacked cell. This again reduces the number of un-attacked cells and number of queens to be placed becomes n−2. Continue doing this, as long as following conditions hold.

- The number of un-attacked cells is not 0.
- The number of queens to be placed is not 0.

If the number of queens to be placed becomes 0, then it's over, we found a solution. But if the number of un-attacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell. We do this recursively. Complete algorithm is given below:

1. Start

For every position col on the same row

If position col is available

- Place the next queen in position col;
- if (row < 4)

putQueen(row+1);

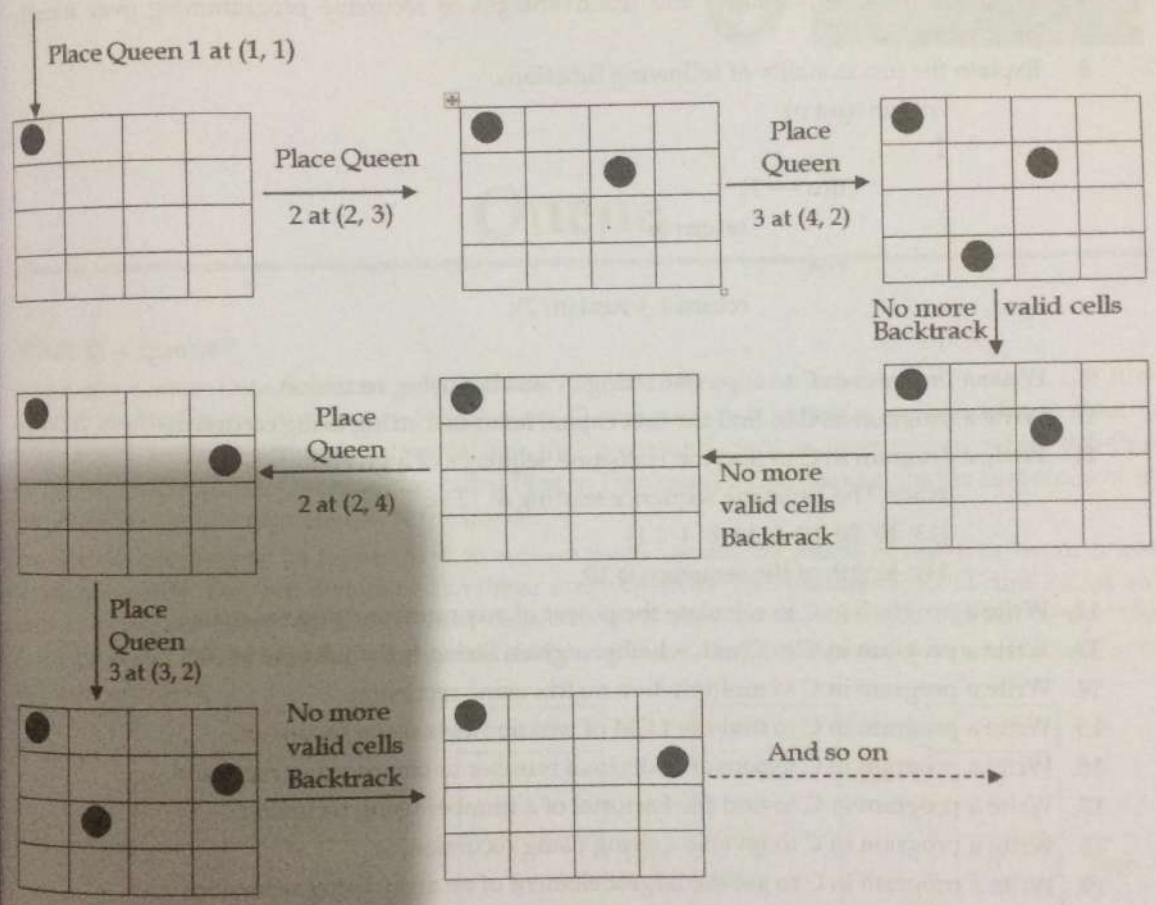
Else

Success;

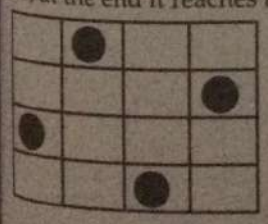
Remove the queen from position col;

2. Stop

Example: Here's how it works for N=4.



So, at the end it reaches the following solution:



So, clearly, the above algorithm, tries solving a sub problem, if that does not result in the solution, it undo whatever changes were made and solve the next sub problem. If the solution does not exists (N=2), then it returns false.

Exercise

1. What is Recursion? What is base condition in recursion?
2. How a particular problem is solved using recursion?
3. Why Stack Overflow error occurs in recursion?
4. What is the difference between direct and indirect recursion?
5. What is difference between tailed and non-tailed recursion?
6. How memory is allocated to different function calls in recursion?
7. What are the disadvantages and disadvantages of recursive programming over iterative programming?
8. Explain the functionality of following functions.

```
int fun1(int n)
{
        if(n == 1)
                return 0;
        else
                return 1 + fun1(n/2);
}
```

9. Write a program in C to copy one string to another using recursion.
10. Write a program in C to find the first capital letter in a string using recursion.
11. Write a program in C to find the Hailstone Sequence of a given number.

    Note: The hailstone sequence starting at 13 is:

    [13 40 20 10 5 16 8 4 2 1]

    The length of the sequence is 10.
12. Write a program in C to calculate the power of any number using recursion.
13. Write a program in C to Check whether a given String is Palindrome or not.
14. Write a program in C to multiply two matrix using recursion.
15. Write a program in C to find the LCM of two numbers using recursion.
16. Write a program in C to convert a decimal number to binary using recursion.
17. Write a program in C to find the Factorial of a number using recursion.
18. Write a program in C to reverse a string using recursion.
19. Write a program in C to get the largest element of an array using recursion.
20. Write a program in C to find GCD of two numbers using recursion.

•••

Chapter

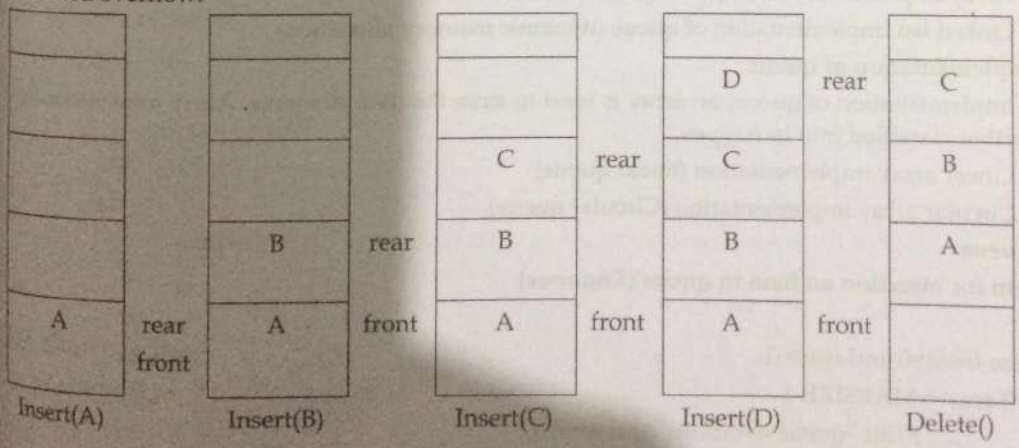
6

Queue

What is a queue?

Queue is a Linear Data Structure which follows First in First out mechanism. It means: the first element inserted is the first one to be removed. Queue uses two variables rear and front. Rear is incremented while inserting an element into the queue and front is incremented while deleting element from the queue. Queue is also called First in First out (FIFO) system since the first element in queue will be the first element out of the queue.

Like stacks, queues may be represented in various ways, usually by means of one-way list or linear arrays. Generally, they are maintained in linear array QUEUE. Two pointers FRONT and REAR are used to represent front and last element respectively. N may be the size of the linear array. The condition when FRONT is NULL indicating that the queue is empty. The condition when REAR is N indicated overflow.



Valid Operations on Queue

- Inserting an element in to the queue
- Deleting an element from the queue
- Displaying the elements in the queue

Applications of Queues

1. Real life examples
   - Waiting in line
   - Waiting on hold for tech support
2. Applications related to Computer Science
   - Threads
   - Task waiting for the printing
   - Job scheduling (e.g. Round-Robin algorithm for CPU allocation)
   - CPU scheduling in Operating system uses Queue. The processes ready to execute and the requests of CPU resources wait in a queue and the request is served on first come first serve basis.
   - **Data buffer -** a physical memory storage which is used to temporarily store data while it is being moved from one place to another is also implemented using Queue.

The Queue as ADT

A queue q of type T is a finite sequence of elements with the operations

- **MakeEmpty(q):** To make q as an empty queue
- **IsEmpty(q):** To check whether the queue q is empty or not. Return true if q is empty, return false otherwise.
- **IsFull(q):** To check whether the queue q is full or not. Return true if q is full, return false otherwise.
- **Enqueue(q, x):** To insert an item x at the rear of the queue, if and only if q is not full.
- **Dequeue(q):** To delete an item from the front of the queue q. if and only if q is not empty.
- **Traverse (q):** To read entire queue that is display the content of the queue.

Implementation of queue

There are two techniques for implementing the queue:

- Array implementation of queue (static memory allocation)
- Linked list implementation of queue (dynamic memory allocation)

Array implementation of queue

In array implementation of queue, an array is used to store the data elements. Array implementation is also further classified into two types

- Linear array implementation (linear queue)
- Circular array implementation (Circular queue)

Linear queue

Algorithm for insertion an item in queue (Enqueue)

1. Start
2. Initialize front=0 and rear=-1

    If rear>=MAXSIZE-1

        Print "queue overflow" and return

    Else

```
        Set, rear=rear+1
        queue[rear]=item
3. Stop
```

Algorithm to delete an element from the queue (Dequeue)

```
1. Start
2. If rear<front
      Print "queue is empty" and return
      Else
            Item=queue [front++]
            Print "item" as a deleted element
2. Stop
```

Declaration of a Queue

```
# define MAXQUEUE 50 /* size of the queue items*/
struct queue
{
      int front;
      int rear;
      int items[MAXQUEUE];
};
typedef struct queue qt;
```

Defining the operations of linear queue

The MakeEmpty function

```
void makeEmpty(qt *q)
{
      q→rear=-1;
      q→front=0;
}
```

The IsEmpty function

```
int IsEmpty(qt *q)
{
      if(q→rear<q→front)
            return 1;
      else
            return 0;
}
```

The Isfull function

```
int IsFull(qt *q)
{
      if(q→rear==MAXQUEUEZIZE-1)
```

```
                return 1;
        else
                return 0;
}
```

The Enqueue function

```
void Enqueue(qt *q, int newitem)
{
        if(IsFull(q))
        {
                printf("queue is full");
                exit(1);
        }
        else
        {
                q→rear++;
                q→items[q→rear]=newitem;
        }
}
```

The Dequeue function

```
int Dequeue(qt *q)
{
        if(IsEmpty(q))
        {
                printf("queue is Empty");
                exit(1);
        }
        else
        {
                return(q→items[q→front]);
                q→front++;
        }
}
```

Menu driven program for array implementation of linear queue in C

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct queue
{
        int item[SIZE];
```

```
        int rear;
        int front;
}
typedef struct queue qu;
void insert(qu*);
void delet(qu*);
void display(qu*);
void main()
{
        int ch;
        qu *q;
        q→rear=-1;
        q→front=0;
        clrscr( );
        printf("Menu for program:\n");
        printf("1:insert\n2:delete\n3:display\n4:exit\n");
        do
        {
                printf("Enter  youer choice\n");
                scanf("%d", &ch);
                switch(ch)
                {
                        case 1:
                                insert(q);
                                break;
                        case 2:
                                delet(q);
                                break;
                        case 3:
                                display( q);
                                break;
                        case 4:
                                exit(1);
                                break;
                        default:
                                printf("Your choice is wrong\n");
                }
        }while(ch<5);
        getch( );
}
/*insert function*/
void insert(qu *q)
```

```
{
        int d;
        printf("Enter  data to be inserted\n");
        scanf("%d",&d);
        if(q→rear==SIZE-1)
        {
                rintf("Queue is full\n");
        }
        else
        {
                q→rear++;
                q→item[q→rear]=d;
        }
}
void delet(qu *q)    /*delete function*/
{
        int d;
        if(q→rear<q→front)
        {
                printf("Queue is empty\n");
        }
        else
        {
                d=q→item[q→front];
                q→front++;
                printf("Deleted item is:");
                printf("%d\n",d);
        }
}
void display(qu *q) /*display function*/
{
        int i;
        if(q→rear<q→front)
        {
                printf("Queue is empty\n");
        }
        else
        {
                for(i=q→front;i<=q→rear;i++)
                {
                        printf("%d\t", q→item[i]) ;
                }
        }
}
```

Output

```
1: Enqueue
2: Dequeue
3: Display
Enter your choice
1
Enter data item to be inserted
99
Enter your choice
1
Enter data item to be inserted
55
Enter your choice
1
Enter data item to be inserted
45
Enter your choice
3
Queue elements are:
99 55 45 Enter your choice
2
Deleted element=99
Enter your choice
2
Deleted element=55
Enter your choice
2
Deleted element=45
Enter your choice
3
Queue empty
Enter your choice
1
Enter data item to be inserted
24
Enter your choice
1
Enter data item to be inserted
56
Enter your choice
3
Queue elements are:
```

24 56 Enter your choice
2
Deleted element=24
Enter your choice

Linked list implementation of linear queue in C

Similar to stack, the queue can also be implemented using both arrays and linked lists. But it also has the same drawback of limited size. Hence, we will be using a linked list to implement the queue. In linked list implementation of queue, we need to create a node for each data items and arrange nodes in first in first out manner.

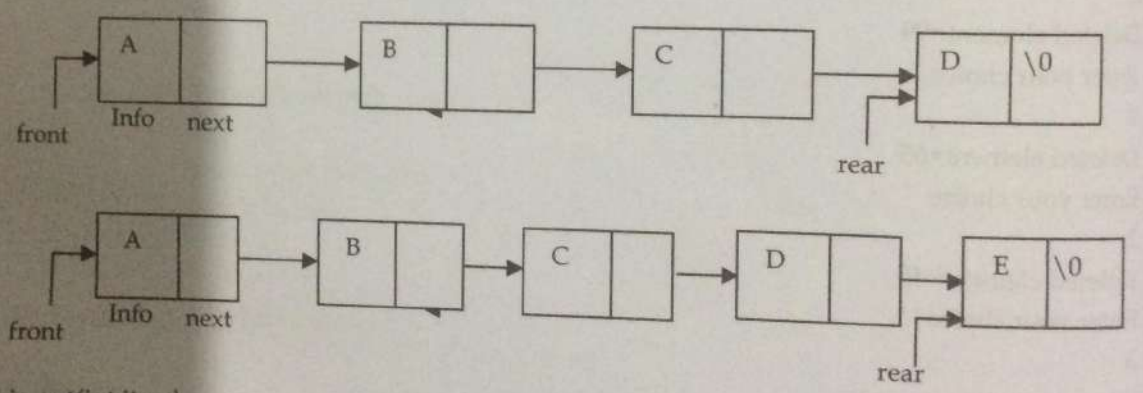
Algorithm

- Create a new node with the value to be inserted.
- If the Queue is empty, then set both front and rear to point to newNode.
- If the Queue is not empty, then set next of rear to the new node and the rear to point to the new node.

The time complexity for Enqueue operation is O(1). The Method for **Enqueue** will be like the following.

Insert function

Let *rear and *front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.

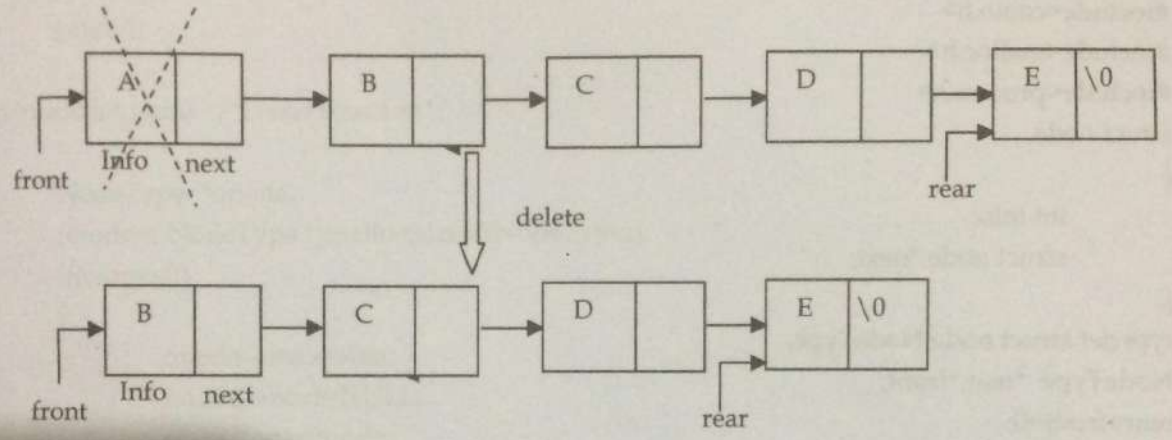



```
void insert(int item)
{
        NodeType *nnode;
        nnode=( NodeType *)malloc(sizeof(NodeType));
        if(rear==0)
        {
                nnode→info=item;
                nnode→next=NULL;
                rear=front=nnode;
        }
        else
        {
```

```
        nnode→info=item;
        nnode→next=NULL;
        rear→next=nnode;
        rear=nnode;
    }
}
```

Delete function

Let *rear and *front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.



Algorithm

- If the Queue is empty, terminate the method.
- If the Queue is not empty, increment front to point to next node.
- Finally, check if the front is null, then set rear to null also. This signifies empty Queue.

The time complexity for Dequeue operation is O(1). The Method for **Dequeue** will be like following.

```
void delet()
{
    NodeType *temp;
    if(front==0)
    {
        printf("Queue contain no elements:\n");
        return;
    }
    else if(front→next==NULL)
    {
        temp=front;
        rear=front=NULL;
        printf("\n Deleted item is %d\n", temp→info);
        free(temp);
    }
    else
```

```
        {
                temp=front;
                front=front→next;
                printf("\nDeleted item is %d\n",temp→info);
                free(temp);
        }
}
```

A Complete C program for linked list implementation of linear queue

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>
struct node
{
        int info;
        struct node *next;
};
typedef struct node NodeType;
NodeType  *rear,*front;
rear=front=0;
void insert(int);
void delet();
void display();
void main()
{
        int choice, item;
        clrscr();
        do
        {
                printf("\n1.Enqueue \n2.Dequeue \n3.Display\n4:Exit\n");
                printf("enter your choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                printf("\nEnter the data:\n");
                                scanf("%d", &item);
                                insert(item);
                                break;
                        case 2:
                                delet( );
                                break;
```

```
                case 3:
                        display();
                        break;
                case 4:
                        exit(1);
                        break;
                default:
                        printf("invalid choice\n");
            }
        }while(choice<5);
        getch();
}
void insert(int item)   /*insert function*/
{
        NodeType  *nnode;
        nnode=( NodeType *)malloc(sizeof(NodeType));
        if(rear==0)
        {
                nnode→info=item;
                nnode→next=NULL;
                rear=front=nnode;
        }
        else
        {
                nnode→info=item;
                nnode→next=NULL;
                rear→next=nnode;
                rear=nnode;
        }
}
void delet()  /*delete function*/
{
        NodeType  *temp;
        if(front==0)
        {
                printf("Queue contain no elements:\n");
                return;
        }
        else if(front→next==NULL)
        {
                temp=front;
                rear=front=NULL;
```

```
                printf("\nDeleted item is %d\n",temp→info);
                free(temp);
        }
        else
        {
                temp=front;
                front=front→next;
                printf("\nDeleted item is %d\n",temp→info);
                free(temp);
        }
}
void display()/*display function*/
{
        NodeType *temp;
        temp=front;
        printf("\nqueue items are:\t");
        while(temp!=NULL)
        {
                printf("%d\t", temp→info);
                temp=temp→next;
        }
}
```

Output

```
1: Enqueue
2: Dequeue
3: Display
4. Exit
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
87
Enter your choice
1
Enter data item to be inserted
55
Enter your choice
3
```

22 87 55 Enter your choice
2
Deleted item=22
Enter your choice
2
Deleted item=87
Enter your choice
2
Deleted item=55
Enter your choice
3
Empty queue
Enter your choice
1
Enter data item to be inserted
43
Enter your choice
3
43 Enter your choice
1
Enter data item to be inserted
54
Enter your choice
3
43 54 Enter your choice
2
Deleted item=43
Enter your choice

Problems in linear queue

The linear arrangement of the queue always considers the elements in forward direction. In the above two algorithms, we had seen that, the pointers front and rear are always incremented as and when we delete or insert element respectively. Suppose in a queue of 10 elements front points to 2th element and rear points to 7th element as follows.

0	1	2	3	4	5	6	7	8	9
		XX	XX	XX	XX	XX			

Queue front rear

When we insert three more elements then the array will become full.

0	1	2	3	4	5	6	7	8	9
		XX	XX	XX	XX	XX	XX	XX	XX

front rear

Later, when we try to insert some elements, then according to the logic when **rear** is n-1 then it encounters an overflow situation. But there are some elements are left blank at the beginning part of the array. Wastage of the space is the main problem with linear queue.

Circular queue

A circular queue is one in which the insertion of a new element is done at very first location of the queue if the last location of the queue is full. When the rear part reaches at the last index of linear queue but front part is not at first index of the queue then in this case if we try to insert some elements, then according to the logic when **rear** is N-1 then it encounters an overflow situation. But there are some elements are left blank at the beginning part of the array. To utilize those left over spaces more efficiently, a circular fashion is implemented in queue representation. The circular fashion of queue reassigns the rear pointer with 0 if it reaches N-1 and beginning elements are free and the process is continued for deletion also. Such queues are called Circular Queue.

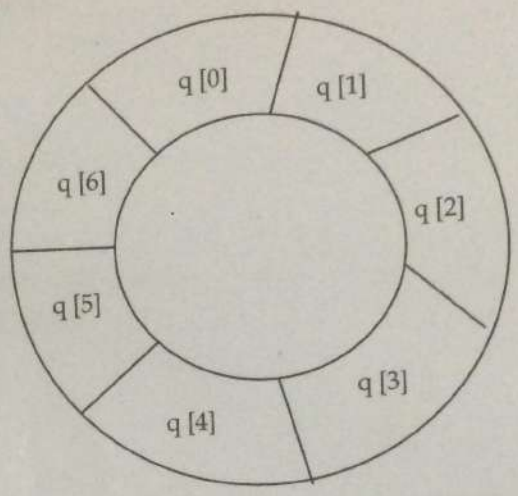


Fig: Circular queue with 7 data items

A circular queue overcomes the problem of unutilized space in linear queue implementation as array. In circular queue we sacrifice one element of the array thus to insert n elements in a circular queue we need an array of size n+1(or we can insert one less than the size of the array in circular queue).

Initialization of Circular queue

rear = front=MAXSIZE-1

Algorithms for inserting an element in a circular queue

This algorithm is assumed that rear and front are initially set to MAZSIZE-1.

1. Check queue full condition as,

```
if (front==(rear+1)%MAXSIZE)
        print Queue is full and exit
else
        rear=(rear+1)%MAXSIZE;  [increment rear by 1]
```

2. cqueue[rear]=item;
3. Stop

Algorithms for deleting an element from a circular queue

This algorithm is assumed that rear and front are initially set to MAZSIZE-1.

1. [Checking empty condition]

```
        if (rear==front)
                Print Queue is empty and exit
        else
                front=(front+1)%MAXSIZE;  [increment front by 1]
```

3. item=cqueue[front];
4. return item;
5. Stop

Declaration of a Circular Queue:

```
# define MAXSIZE 50 /* size of the circular queue items*/
struct cqueue
{
        int front;
        int rear;
        int items[MAXSIZE];
};
typedef struct cqueue cq;
```

The IsEmpty function

```
int IsEmpty()
{
        if(rear==front)
                return 1;
        else
                return 0;
}
```

The Isfull function

```
int IsFull()
{
        if (front==(rear+1)%SIZE)
                return 1;
        else
                return 0;
}
```

The Enqueue function

```
void Enqueue(cq *q, int newitem)
{
        if(q→front==(q→rear+1)%MAXSIZE)
        {
```

```
                printf("queue is full");
                exit(1);
        }
        else
        {
                q→rear=(q→rear+1)%MAXSIZE;
                q→items[q→rear]=newitem;
        }
}
```

The Dequeue function

```
int Dequeue(cq *q)
{
        if(q→rear<q→front)
        {
                printf("queue is Empty");
                exit(1);
        }
        else
        {
                q→front=(q→front+1)%MAXSIZE;
                return(q→items[q→front]);
        }
}
```

Array implementation of circular queue with scarifying one cell in C

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
struct cqueue
{
        int item[SIZE];
        int rear;
        int front;
};
typedef struct cqueue qu;
void insert(qu*);
void delet(qu*);
void display(qu*);
void main()
{
        int ch;
        qu *q;
```

```
    q→rear=SIZE-1;
    q→front=SIZE-1;
    clrscr();
    printf("Menu for program:\n");
    printf("1:Enqueue\n2:Dequeue\n3:Display\n4:exit\n");
    do
    {
            printf("Enter your choice\n");
            scanf("%d",&ch);
            switch(ch)
            {
                    case 1:
                            insert(q);
                            break;
                    case 2:
                            delet(q);
                            break;
                    case 3:
                            display(q);
                            break;
                    case 4:
                            exit(1);
                            break;
                    default:
                            printf("Your choice is wrong\n");
            }
    }while(ch<5);
    getch();
}
/*insert function*/
void insert(qu *q)
{
    int d;
    if((q→rear+1)%SIZE==q→front)
    printf("Queue is full\n");
    else
    {
            q→rear=(q→rear+1)%SIZE;
            printf ("Enter data to be inserted\n");
            scanf("%d",&d);
            q→item[q→rear]=d;
    }
```

```
}
/*delete function*/
void delet(qu *q)
{
        if(q→rear==q→front)
        printf("Queue is empty\n");
        else
        {
                q→front=(q→front+1)%SIZE;
                printf("Deleted item is:");
                printf("%d\n",q→item[q→front]);
        }
}
/*display function*/
void display(qu *q)
{
        int i;
        if(q→rear==q→front)
        printf("Queue is empty\n");
        else
        {
                printf("Items of queue are:\n");
                for(i=(q→front+1)%SIZE;i!=q→rear;i=(i+1)%SIZE)
                {
                        printf("%d\t",q→item[i]);
                }
                printf("%d\t",q→item[q→rear]);
        }
}
```

Output

```
1: Enqueue
2: Dequeue
3: Display
4: Exit
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
54
```

```
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
77
Enter your choice
1
Enter data item to be inserted
99
Queue full
Enter your choice
3
Queue elements are:
22 54 22 77 Enter your choice
2
Deleted element=22
Enter your choice
2
Deleted element=54
Enter your choice
2
Deleted element=22
Enter your choice
1
Enter data item to be inserted
43
Enter your choice
3
Queue elements are:
77 43 Enter your choice
```

Array Implementation of circular queue without sacrificing one cell by using a count variable

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
struct cqueue
{
        int item[SIZE];
        int rear;
```

```
        int front;
};
int count=0;
typedef struct cqueue qu;
void insert(qu*);
void delet(qu*);
void display(qu*);
void main( )
{
        int ch;
        qu *q;
        q→rear=SIZE-1;
        q→front=SIZE-1;
        clrscr( );
        printf("Menu for program:\n");
        printf("1:Enqueue\n2:Dequeue\n3:Display\n4:exit\n");
        do
        {
                printf("Enter youer choice\n");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:
                                insert(q);
                                break;
                        case 2:
                                delet(q);
                                break;
                        case 3:
                                display(q);
                                break;
                        case 4:
                                exit(1);
                                break;
                        default:
                                printf("Your choice is wrong\n");
                }
        }while(ch<5);
        getch( );
}
/*insert function*/
void insert(qu *q)
```

```
{
        int d;
        if(count==SIZE)
                printf("Queue is full\n");
        else
        {
                q→rear=(q→rear+1)%SIZE;
                printf ("Enter data to be inserted\n");
                scanf("%d",&d);
                q→item[q→rear]=d;
                count++;
        }
}
/*delete function*/
void delet(qu *q)
{
        if(count==0)
                printf("Queue is empty\n");
        else
        {
                q→front=(q→front+1)%SIZE;
                printf("Deleted item is:");
                printf("%d\n",q→item[q→front]);
                count--;
        }
}
/*display function*/
void display(qu *q)
{
        int i;
        if(q→rear==q→front)
                printf("Queue is empty\n");
        else
        {
                printf("Items of queue are:\n");
                for(i=(q→front+1)%SIZE; i!=q→rear; i=(i +1)%SIZE)
                {
                        printf("%d\t",q→item[i]);
                }
                printf("%d\t",q→item[q→rear]);
        }
}
```

Output

```
1: Enqueue
2: Dequeue
3: Display
4: Exit
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
44
Enter your choice
1
Enter data item to be inserted
23
Enter your choice
1
Enter data item to be inserted
76
Enter your choice
1
Enter data item to be inserted
16
Enter your choice
1
Enter data item to be inserted
22
Queue full
Enter your choice
3
Queue elements are:
22 44 23 76 16 Enter your choice
2
Deleted element=22
Enter your choice
2
Deleted element=44
Enter your choice
2
Deleted element=23
```

Enter your choice

3

Queue elements are:

76 16 Enter your choice

Linked list implementation of circular queue in C

It is easier to represent a queue as a circular list than as a linear list. As a linear list a queue is specified by two pointers, one to the front of the list and the other to its rear. However, by using a circular list, a queue may be specified by a single pointer q to that list.

Insertion function

```
void insert(int item)
{
        NodeType *nnode;
        nnode=( NodeType *)malloc(sizeof(NodeType));
        nnode→info=item;
        if(pq==NULL)
        pq=nnode;
        else
        {
                nnode→next=pq→next;
                pq→next=nnode;
                pq=nnode;
        }
}
```

Deletion function

```
void delet(int item)
{
        NodeType *temp;
        if(pq==NULL)
        {
                printf("void deletion\n");
                exit(1);
        }
        else if(pq→next==pq)    //for only one node
        {
                printf("poped item=%d", pq→info);
                pq=NULL;
        }
        else
        {
                temp=pq→next;
                pq→next=temp→next;
```

```
            printf("poped item=%d", temp→info);
            free(temp);
        }
}
```

Priority queue

A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

The best application of priority queue is observed in CPU scheduling. A short job is given higher priority over the longer one. Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

In normal queue data structure, insertion is performed at the end of the queue and deletion is performed based on the FIFO principle. This queue implementation may not be suitable for all situations.

Consider a networking application where server has to respond for requests from multiple clients using queue data structure. Assume four requests arrived to the queue in the order of R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. Queue is as follows:

R1:20	R2:2	R3:10	R4:5					

front rear

Now, check waiting time for each request to be complete.

1. R1: 20 units of time
2. R2: 22 units of time (R2 must wait till R1 complete - 20 units and R2 itself requires 2 units. Total 22 units)
3. R3: 32 units of time (R3 must wait till R2 complete - 22 units and R3 itself requires 10 units. Total 32 units)
4. R4: 37 units of time (R4 must wait till R3 complete - 35 units and R4 itself requires 5 units. Total 37 units)

Here, average waiting time for all requests (R1, R2, R3 and R4) is (20+22+32+37)/4 ≈ 27 units of time.

That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is 27 units of time.

Now, consider another way of serving these requests. If we serve according to their required amount of time. That means, first we serve R2 which has minimum time required (2) then serve R4 which has second minimum time required (5) then serve R3 which has third minimum time required (10) and finally R1 which has maximum time required (20).

Now, check waiting time for each request to be complete.

1. R2: 2 units of time
2. R4: 7 units of time (R4 must wait till R2 complete 2 units and R4 itself requires 5 units. Total 7 units)

3. R3: 17 units of time (R3 must wait till R4 complete 7 units and R3 itself requires 10 units. Total 17 units)
4. R1: 37 units of time (R1 must wait till R3 complete 17 units and R1 itself requires 20 units. Total 37 units)

Here, average waiting time for all requests (R1, R2, R3 and R4) is (2+7+17+37)/4 ≈ 15 units of time.

From above two situations, it is very clear that, by using second method server can complete all four requests with very less time compared to the first method. This is what exactly done by the priority queue.

Types of priority queues

Descending priority queue (max priority queue)

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

The following are the operations performed in a Max priority queue

1. isEmpty() - Check whether queue is Empty.
2. insert() - Inserts a new value into the queue.
3. findMax() - Find maximum value in the queue.
4. remove() - Delete maximum value from the queue.

Ascending priority queue (min priority queue)

Min Priority Queue is similar to max priority queue except removing maximum element first, we remove minimum element first in min priority queue. The following operations are performed in Min Priority Queue

1. **isEmpty()** - Check whether queue is Empty.
2. **insert()** - Inserts a new value into the queue.
3. **findMin()** - Find minimum value in the queue.
4. **remove()** - Delete minimum value from the queue.

Declaration

```
#define MAXQUEUE 10 /* size of the queue items*/
struct pqueue
{
        int front;
        int rear;
        int items[MAXQUEUE];
};
struct pqueue *pq;
```

The priority queue ADT

An ascending priority queue of elements of type T is a finite sequence of elements of T together with the operations:

- **MakeEmpty(p):** Create an empty priority queue p
- **Empty (p):** Decide whether the priority queue p is empty or not
- **Insert (p, x):** Add element x on the priority queue p

- **DeleteMin(p):** If the priority queue p is not empty, remove the minimum element of the queue and return it.
- **FindMin(p):** Retrieve the minimum element of the priority queue p.

Array Implementation of ascending priority queue in C

Here a one dimensional array is used to store the elements and elements are stored in given sequence of data input but they are deleted in ascending order i.e. delete smallest element first. This means giving priority to shortest job first.

Example: Implementation of ascending priority queue

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct cqueue
{
        int item[SIZE];
        int rear;
        int front;
};
typedef struct queue pq;
void insert(pq*);
void delet(pq*);
void display(pq*);
void main( )
{
        int ch;
        pq *q;
        q→rear=-1;
        q→front=0;
        clrscr( );
        printf("Menu for program:\n");
        printf("1:Enqueue\n2:Dequeue\n3:Display\n4:Exit\n");
        do
        {
                printf("Enter youer choice\n");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:
                                insert(q);
                                break;
                        case 2:
                                delet(q);
                                break;
```

```
                case 3:
                        display(q);
                        break;
                case 4:
                        exit(1);
                        break;
                default:
                        printf("Your choice is wrong\n");
        }
    }while(ch<5);
    getch( );
}
/*insert function*/
void insert(pq  *q)
{
    int d;
    if(q→rear==SIZE-1)
        printf("Queue is full\n");
    else
    {
        printf ("Enter data to be inserted\n");
        scanf("%d",&d);
        q→rear++;
        q→item[q→rear]=d;
    }
}
/*delete function*/
void delet(pq  *q)
{
    int i, temp=0, x;
    x=q→item[q→front];
    if(q→rear<q→front)
    {
        printf("Queue is empty\n");
        return 0;
    }
    else
    {
        for(i=q→front+1; i<q→rear; i++)
        {
            if(x>q→item[i])
            {
```

```
                        temp=i;
                        x=q→item[i];
                }
        }
        for(i=temp; i< q→rear-1; i++)
        {
                q→item[i]=q→item[i+1];
        }
        q→rear–;
        return x;
    }
}
/*display function*/
void display(pq *q)
{
        int i;
        if(q→rear < q→front)
                printf("Queue is empty\n");
        else
        {
                printf("Items of queue are:\n");
                for(i=q→front ; i<=q→rear; i++)
                {
                        printf("%d\t", q→item[i]);
                }
        }
}
```

Output

```
1: Enqueue
2: Dequeue
3: Display
4: Exit
Enter your choice
1
Enter data item to be inserted
99
Enter your choice
1
Enter data item to be inserted
76
Enter your choice
1
Enter data item to be inserted
```

```
34
Enter your choice
3
Queue elements are:
99 76 34 Enter your choice
2
Deleted element is= 34
Enter your choice
3
Queue elements are:
99 76 Enter your choice
1
Enter data item to be inserted
23
Enter your choice
3
Queue elements are:
99 76 23 Enter your choice
1
Enter data item to be inserted
232
Enter your choice
3
Queue elements are:
99 76 23 232 Enter your choice
2
Deleted element is= 23
Enter your choice
2
Deleted element is= 76
```

Exercise

1. What is queue? What are main drawbacks of linear queue over circular queue?
2. What is concept behind priority queue? Write complete java program to implement ascending priority queue.
3. What are the application areas of queue? Show that queue as ADT.
4. How can you implement circular queue in java by using linked list?
5. What are the application areas of priority queue? Show that priority queue act as an ADT.
6. What are the basic operations of queue? Describe them with suitable practical example.
7. What is queue? Differentiate between simple queue and circular queue.
8. What is FIFO? How it is differ from LIFO?
9. What is a Dequeue? Explain structure of queue with suitable example.

10. Which Data Structure should be used for implementing LRU cache?
11. What are the full and empty conditions for circular queue?
12. Queue is a default class in java. Is this argument is true or not? Justify.
13. In which case circular queue is more superior than linear queue?
14. Priority queue violets the queue property. Is this argument is true? Justify.
15. Write complete program in C to implement various operations of linear queue.
16. How circular queue differ from linear queue? Explain.
17. Define ascending and descending priority queue with suitable example.
18. What are the main operations of priority queue? Explain.
19. Describe advantages and disadvantages of circular queue.
20. What are the application areas of queue in real life? Explain.

•••

Chapter

7

Tree

Introduction

A tree is a nonlinear data structure that models a hierarchical organization. The characteristic features are that each element may have several successors called its children and every element except one called the root has a unique predecessor called its parent. A tree is a data structure which consists of a finite set of elements, called nodes and a finite set of directed lines, called branches that connect the nodes. The number of branches associated with a node is the degree of the node. The branch coming towards the node is the in-degree of the node. Similarly the outgoing branches from the node mean the out-degree of that node. The first node of the tree is known as root. When the tree is empty, then root is equals to NULL. In another case the in-degree of the root is always 0.

Trees can be defined in two ways: non-recursively and recursively. The non-recursive definition is the more direct technique, so we begin with it. The recursive formulation allows us to write simple algorithms to manipulate trees. Trees fall into the category of non-primitive non-linear data structures in the classification of data structure. They contain a finite set of data items referred as nodes. We can represent a hierarchical relationship between the data elements using trees.

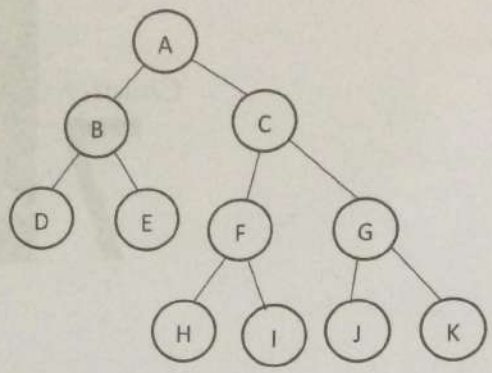
A Tree has the following characteristics:

- The top item in a hierarchy of a tree is referred as the root of the tree.
- The remaining data elements are partitioned into a number of mutually exclusive subsets and they itself a tree and are known as the sub-tree.
- Unlike natural trees, trees in the data structure always grow in length towards the bottom.

Definition

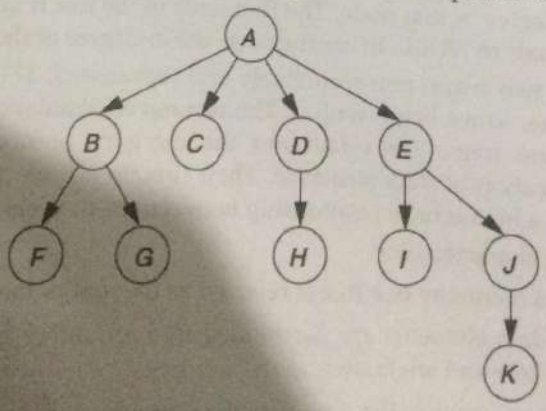
A tree is a nonlinear data structure and is generally defined as a nonempty finite set of elements which consists of set of nodes called vertices and set of edges which links vertices such that:

- T contains a distinguished node called root of the tree.
- The remaining elements of tree form an ordered collection of zero or more disjoint subsets called sub tree.



Terminology

- **Root Node:** The starting node of a tree is called Root node of that tree
- **Terminal Nodes:** The node which has no children is said to be terminal node or leaf.
- **Non-Terminal Node:** The nodes which have children is said to be Non-Terminal Nodes
- **Degree:** The degree of a node is number of sub trees of that node
- **Levels:** Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Depth:** The length of largest path from root to terminals is said to be depth of tree.
- **Depth of a node:** The **depth of a node** is the length of the path from the root to the node.
- **Height of a node:** The **height of a node** is the length of the path from the node to the deepest leaf.
- **Siblings:** The children of same parent are said to be siblings
- **Ancestors:** The ancestors of a node are all the nodes along the path from the root to the node
- **Keys:** Key represents a value of a node based on which a search operation is to be carried out for a node.
- **Sub-tree:** Sub-tree represents the descendants of a node.
- **Traversing**: Traversing means passing through nodes in a specific order.



The root node is A; A's children are B, C, D, and E. Because A is the root, it has no parent; all other nodes have parents. For instance, B's parent is A. A node that has no children is called a leaf. The leaves in this tree are C, F, G, H, I, and K. The length of the path from A to K is 3 (edges); the length of the path from A to A is 0 (edges).

A tree with N nodes must have N – 1 edges because every node except the parent has an incoming edge. The depth of a node in a tree is the length of the path from the root to the node. Thus the depth of the root is always 0, and the depth of any node is 1 more than the depth of its parent. The height of a node in a tree is the length of the path from the node to the deepest leaf. Thus the height of E is 2. The height of any node is 1 more than the height of its maximum-height child. Thus the height of a tree is the height of the root. Nodes with the same parent are called siblings; thus B, C, D, and E are all siblings. If there is a path from node u to node v, then u is an ancestor of v and v is a descendant of u. If u ≠ v, then u is a proper ancestor of v and v is a proper descendant of u.

Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

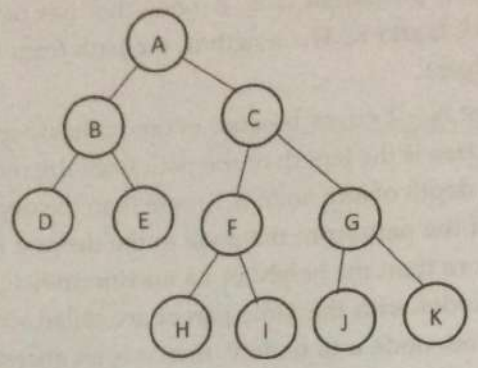
Binary Trees

A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the **root** of the tree. The other two subsets are themselves binary trees called the **left** and **right sub-trees** of the original tree. A left or right sub tree can be empty. Each element of a binary tree is called a **node** of the tree.

Properties of Binary tree

- Trees are used to represent data in hierarchical form.
- Binary tree is the one in which each node has maximum of two child- node.
- The order of binary tree is '2'.
- Binary tree does not allow duplicate values.
- The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}
- Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$
- A Binary Tree with L leaves has at least $\lceil Log2L \rfloor + 1$ levels

The following figure shows a binary tree with 11 nodes where A is the root.



Simply, a tree with at most two children for every internal node is called binary tree.

Advantages of Binary Tree:

- Searching in Binary tree become faster
- Maximum and minimum elements can be directly picked up
- It is used for graph traversal and to convert an expression to postfix and prefix forms

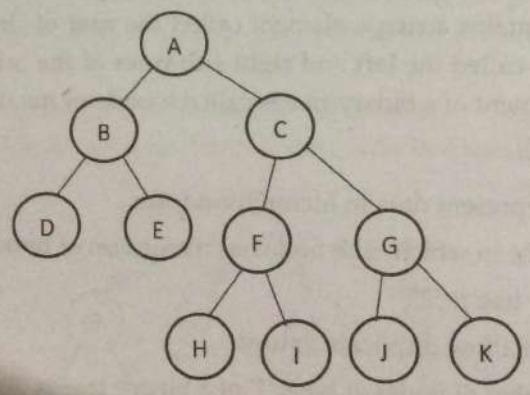
Types of binary tree

Binary tree can be categorized into following 3 types

- Strictly binary tree
- Complete binary tree
- Almost complete binary tree

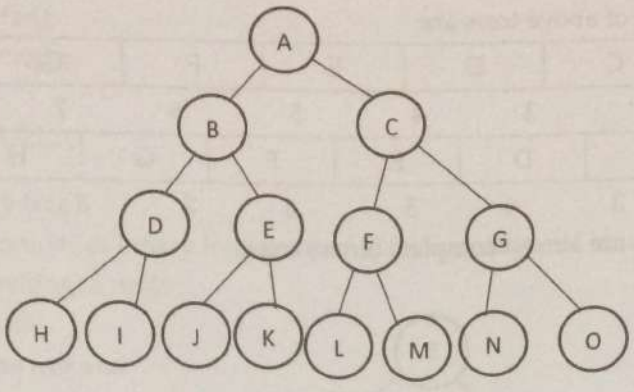
Strictly binary tree

If every non-leaf node in a **binary tree** has nonempty left and right sub-trees, the **tree** is termed a **strictly binary tree**. Or, to put it another way, all of the nodes in a **strictly binary tree** are of degree zero or two, never degree one. A **strictly binary tree** with n leaves always contains (2n – 1) nodes. Simply, if every non-leaf (internal) node in a binary tree has nonempty left and right sub-trees, then such a tree is called a **strictly binary tree.**



Complete binary tree

A complete binary tree of depth d is called strictly binary tree if all of whose leaves are at level d. A complete binary tree with depth d has 2^d leaves and (2^d-1) non-leaf (internal) nodes.



Properties of Full Binary Tree

1. A binary tree of height h with no missing node.
2. All leaves are at level h and all other nodes have two children.
3. All the nodes that are at a level less than h have two children each.
4. If a full binary tree has i internal nodes:
   - Number of leaves l = i+1
   - Total number of nodes n = 2*i+1
5. If a full binary tree has n nodes:
   - Number of internal nodes i = (n-1)/2
   - Number of leaves l=(n+1)/2
6. If a full binary tree has l leaves:
   - Total Number of nodes n=2*l-1

Almost complete binary tree

A binary tree of depth **d** is an almost complete binary tree if:

- Each leaf in the tree is either at level **d** or at level **d** – 1.
- For any node **nd** in the tree with a right descendant at level **d**, all the left descendants of **nd** that are leaves are also at level **d**. It means nodes should be present in left to right at any level; there should not be any missing node in the traversal.

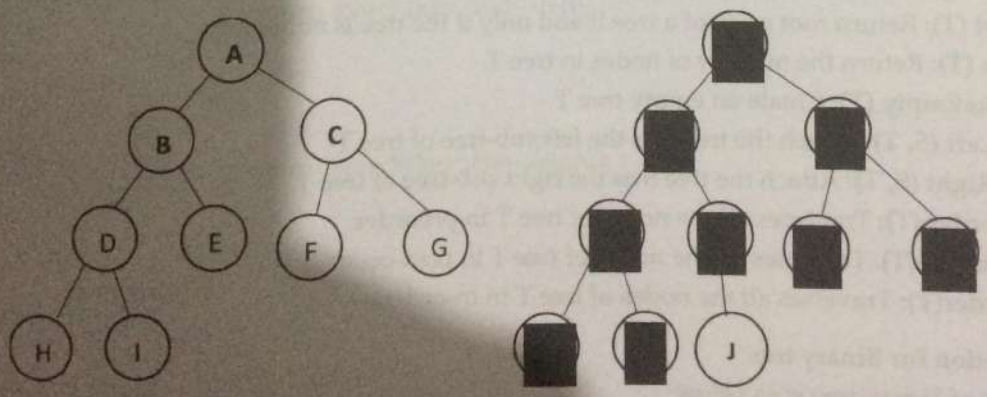


Fig: Almost complete binary trees

Simply, a binary tree is called almost complete binary tree if their array representation has not any null values within the allocated array.

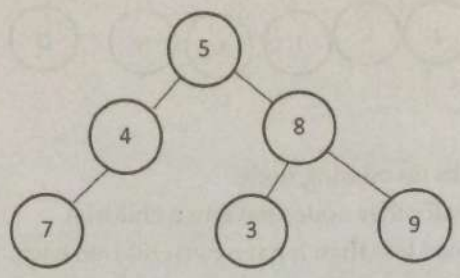
The array representations of above trees are:

A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

Hence both of above trees are almost complete binary trees.

Let's take a binary tree as,



Its array representation is:

5	4	8	7	Null	3	9
0	1	2	3	4	5	6

Since array representation of above binary tree contains null value within the array, hence the above binary tree is not almost complete binary tree.

Operations on Binary tree

- **Father (n, T):** Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- **LeftChild (n, T):** Return the left child of node n in tree T. Return NULL if n does not have a left child.
- **RightChild (n, T):** Return the right child of node n in tree T. Return NULL if n does not have a right child.
- **Info (n, T):** Return information stored in node n of tree T (i.e. Content of a node).
- **Sibling (n, T):** return the sibling node of node **n** in tree T. Return NULL if n has no sibling.
- **Root (T):** Return root node of a tree if and only if the tree is nonempty.
- **Size (T):** Return the number of nodes in tree T
- **MakeEmpty (T):** Create an empty tree T
- **SetLeft (S, T):** Attach the tree S as the left sub-tree of tree T
- **SetRight (S, T):** Attach the tree S as the right sub-tree of tree T.
- **Preorder (T):** Traverses all the nodes of tree T in preorder.
- **postorder(T):** Traverses all the nodes of tree T in post-order
- **Inorder(T):** Traverses all the nodes of tree T in in-order.

C representation for Binary tree

The structure of binary tree is as below:

```
struct bnode
{
        int info;
```

```
    struct bnode *left;
    struct bnode *right;
};
struct bnode *root=NULL;
```

Operations of binary tree

There are a lot of operations of binary tree; some of main operations of binary tree are listed below:

- Finding (Searching) a node
- Inserting a node
- Traversing the tree and
- Deleting a node.

Finding a Node

Finding a node with a specific key is the simplest of the major tree operations, so let's start with that. Remember that the nodes in a binary search tree correspond to objects containing information. They could be person objects, with an employee number as the key and also perhaps name, address, telephone number, salary, and other fields. Or they could represent car parts, with a part number as the key value and fields for quantity on hand, price, and so on. However, the only characteristics of each node that we can see in the Workshop applet are a number and a color. A node is created with these two characteristics and keeps them throughout its life.

```
struct node *search(int key, struct node *leaf)
{
        if( leaf != 0 )
        {
                if(key==leaf→key_value)
                {
                        return leaf;
                }
                else if(key<leaf→key_value)
                {
                        return search(key, leaf→left);
                }
                else
                {
                        return search(key, leaf→right);
                }
        }
        else return 0;
}
```

Inserting a Node

To insert a node we must first find the place to insert it. This is much the same process as trying to find a node that turns out not to exist, as described in the section on Find. We follow the path from the root to the appropriate node, which will be the parent of the new node. Once this parent is found,

the new node is connected as its left or right child, depending on whether the new node's key is less than or greater than that of the parent.

Binary Tree Traversal Techniques

Traversing a tree means visiting each node in a specified order. This process is not as commonly used as finding, inserting, and deleting nodes. One reason for this is that traversal is not particularly fast. But traversing a tree is useful in some circumstances and the algorithm is interesting. The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner. There are three popular methods of traversal:

- Pre-order traversal
- In-order traversal
- Post-order traversal

The order most commonly used for binary search trees is in-order.

Pre-order traversal

In preorder traversing first **root** is visited followed by **left** sub-tree and **right** sub-tree.

The preorder traversal of a nonempty binary tree is defined as follows:

- Visit the root node
- Traverse the left sub-tree in preorder
- Traverse the right sub-tree in preorder

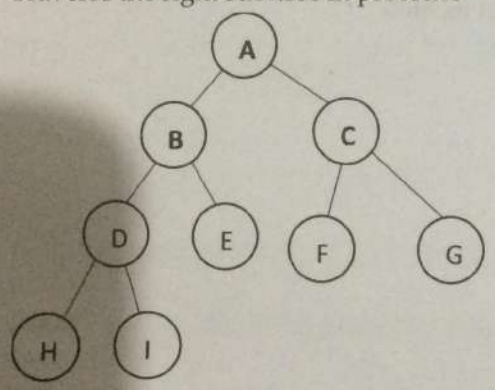


Fig: - Binary tree

The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

C function for preorder traversing

```
void preorder( struct bnode *root)
{
        if(root!=NULL)
        {
                printf ("%c", root→info);
                preorder (root→left);
                preorder (root→right);
        }
}
```

In-order traversal

The in-order traversal of a nonempty binary tree is defined as follows:

- Traverse the left sub-tree in in-order
- Visit the root node
- Traverse the right sub-tree in in-order

The in-order traversal output of the given tree is: H D I B E A F C G

C function for in-order traversing

```
void inorder( struct bnode *root)
{
        if(root!=NULL)
        {
                inorder(root→left);
                printf("%c", root→info);
                inorder(root→right);
        }
}
```

Post-order traversal

The post-order traversal of a nonempty binary tree is defined as follows:

- Traverse the left sub-tree in post-order
- Traverse the right sub-tree in post-order
- Visit the root node

The post-order traversal output of the given tree is: H I D E B F G C A

C function for post-order traversing

```
void post-order( struct bnode *root)
{
        if(root!=NULL)
        {
                post-order(root→left);
                post-order(root→right);
                printf("%c", root→info);
        }
}
```

Complete menu driven java program to perform various operations in Binary tree

```
#include<stdio.h>
#include<alloc.h>
#include<conio.h>
#include<stdio.h>
struct tree
{
```

```
        int info;
        struct tree *left;
        struct tree *right;
};
struct tree *insert(struct tree *, int);
void inorder(struct tree *);
void postorder(struct tree *);
void preorder(struct tree *);
struct tree *delet(struct tree *, int);
struct tree *search(struct tree *);
void main()
{
        struct tree *root;
        int choice, item,item_no;
        root = NULL;
        clrscr();
        do {
                do {
                        printf("\n 1. Insert in Binary Tree ");
                        printf("\n 2. Delete from Binary Tree ");
                        printf("\n 3. In order traversal of Binary tree");
                        printf("\n 4. Post order traversal of Binary tree");
                        printf("\n 5. Preorder traversal of Binary tree");
                        printf("\n 6. Search and replace ");
                        printf("\n 7. Exit ");
                        printf("\n Enter choice : ");
                        scanf(" %d", &choice);
                        if(choice<1 || choice>7)
                        printf("\n Invalid choice - try again");
                }
                while (choice<1 || choice>7);
                switch(choice)
                {
                        case 1:
                                printf("\n Enter new element: ");
                                scanf("%d", &item);
                                root= insert(root, item);
                                printf("\n root is %d", root→info);
                                printf("\n In order traversal of binary tree is : ");
                                inorder(root);
                                break;
                        case 2:
                                printf("\n Enter the element to be deleted : ");
```

```
                scanf(" %d", &item_no);
                root=delet(root, item_no);
                inorder(root);
                break;
        case 3:
                printf("\n Inorder traversal of binary tree is : ");
                inorder(root);
                break;
        case 4:
                printf("\n Postorder traversal of binary tree is : ");
                postorder(root);
                break;
        case 5:
                 printf("\n Preorder traversal of binary tree is : ");
                preorder(root);
                break;
        case 6:
        printf("\n Search and replace operation in binary tree ");
                root=search(root);
                break;
        default:
                 printf("\n End of program ");
            }
    }
    while(choice !=7);
}

struct tree *insert(struct tree *root, int x)
{
    if(!root)
    {
        root=(struct tree*)malloc(sizeof(struct tree));
        root→info = x;
        root→left = NULL;
        root→right = NULL;
        return(root);
    }
    if(root→info > x)
        root→left = insert(root→left, x);
    else
    {
        if(root→info < x)
            root→right = insert(root→right, x);
```

```
        }
        return(root);
}
void inorder(struct tree *root)
{
        if(root != NULL)
        {
                inorder(root→left);
                printf(" %d",root→info);
                inorder(root→right);
        }
        return;
}
void postorder(struct tree *root)
{
        if(root != NULL)
        {
                postorder(root→left);
                postorder(root→right);
                printf(" %d", root→info);
        }
        return;
}
void preorder(struct tree *root)
{
        if(root != NULL)
        {
                printf(" %d",root→info);
                preorder(root→left);
                preorder(root→right);
        }
        return;
}
struct tree *delet(struct tree *ptr, int x)
{
        struct tree *p1,*p2;
        if(!ptr)
        {
                printf("\n Node not found ");
                return(ptr);
        }
        else
        {
```

```
if(ptr→info < x)
{
        ptr→right = delet(ptr→right,x);
}
else if (ptr→info >x)
{
        ptr→left=delet(ptr→left,x);
        return ptr;
}
else    /* no. 2 else */
{
        if(ptr→info == x)  /* no. 2 if */
        {
                if(ptr→left == ptr→right) /* A leaf node*/
                {
                        free(ptr);
                        return(NULL);
                }
                else if(ptr→left==NULL)  /* a right subtree */
                {
                        p1=ptr→right;
                        free(ptr);
                        return p1;
                } else if(ptr→right==NULL)   /* a left subtree */
                {
                        p1=ptr→left;
                        free(ptr);
                        return p1;
                }
                else
                {
                        p1=ptr→right;
                        p2=ptr→right;
                        while(p1→left != NULL)
                                p1=p1→left;
                        p1→left=ptr→left;
                        free(ptr);
                        return p2;
                }
        }
    }
}
return(ptr);
```

```
}
/* function to search and replace an element in the binary tree */
struct tree *search(struct tree *root)
{
        int nkey, i, key;
        struct tree *ptr;
        ptr=root;
        printf("\n Enter the element to be searched :");
        scanf(" %d",&key);
        while(ptr)
        {
                if(key>ptr→info)
                        ptr=ptr→right;
                else if(key<ptr→info)
                        ptr=ptr→left;
                else
                   break;
        }
        if(ptr)
        {
                printf("\n Element %d which was searched is found and is =%d", key,
                        ptr→info);
                printf("\n Do you want replace it, press 1 for yes : ");
                scanf(" %d", &i);
                if(i==1)
                {
                        printf("\n Enter new element :");
                        scanf(" %d",& nkey);
                        ptr→info= nkey;
                }
                else
                  printf("\n It's okay");
        }
        else
                printf("\n Element %d does not exist in the binary tree", key);
        return(root);
}
```

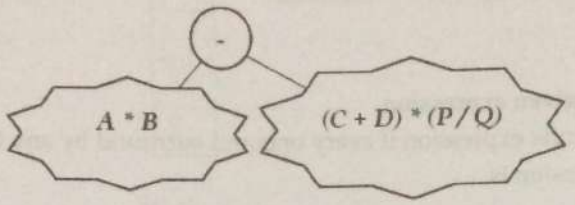
Expression tree

A binary tree with each leaf node contains the operands and internal node contains the operators of given expression is called expression tree. During constructing expression tree from given expression we take an operator with lowest precedence as root node and set left sub tree and right sub tree around the root node. Again left sub tree and right sub tree acts as expression tree recursively.

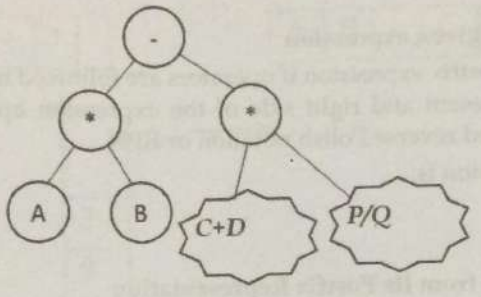
Example: Construct an expression tree of given infix expression

A * B - (C + D) * (P / Q)

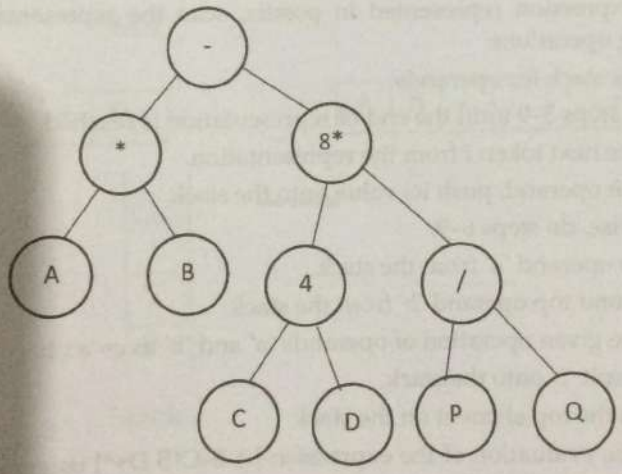
Step 1:



Step 2:



Step 3:



The expression tree for a given expression can be built recursively from the following rules:

- The expression tree for a single operand is a single root node that contains it.
- If E_1 and E_2 are expressions represented by expression trees T_1 and T_2 and if **op** is an operator, then the expression tree for the expression E_1 op E_2 is the tree with root node containing op and sub-trees T_1 and T_2.

An expression has three representations, depending upon which traversal algorithm is used to traverse its tree. The preorder traversal produces the prefix representation, the in-order traversal produces the infix representation, and the post-order traversal produces the postfix representation of the expression. The postfix representation is also called reverse Polish notation or RPN.

Prefix representation of given expression

An expression is called prefix expression if operands are followed by operators. This means towards left side operators are present and right side of the expression operands are present.

Example: A prefix expression is,

+*-a b c/6+b d

Infix representation of given expression

An expression is called infix expression if every operand surround by any two operands.

Example: An infix expression is,

a-b*c+6/b+d

Postfix representation of given expression

An expression is called postfix expression if operators are followed by operands. This means towards left side operands are present and right side of the expression operators are present. The postfix representation is also called reverse Polish notation or RPN.

Example: A prefix expression is,

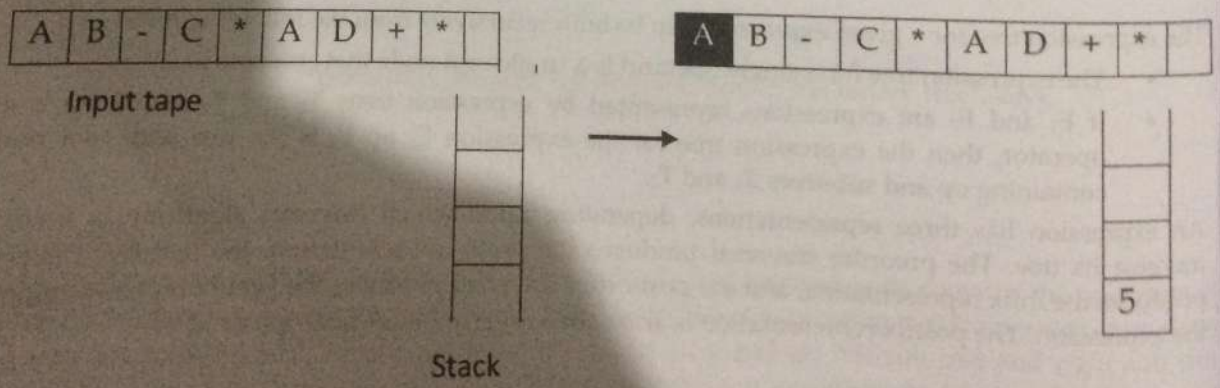
a b-c*6 a d+/+

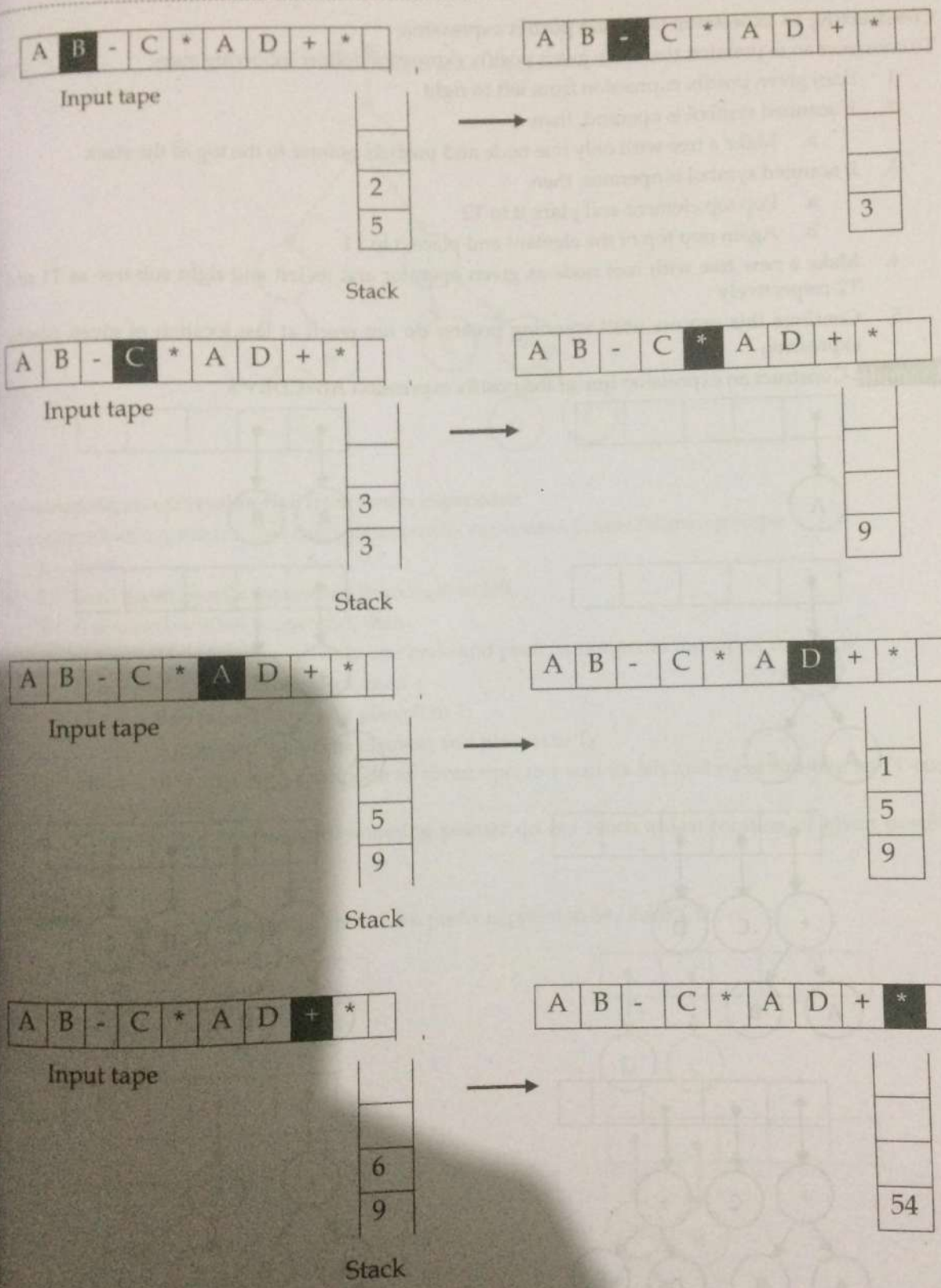
Evaluating an Expression from Its Postfix Representation

To evaluate an expression represented in postfix, scan the representation from left to right and perform following operations:

1. Create a stack for operands.
2. Repeat steps 3–9 until the end of representation is reached.
3. Read the next token *t* from the representation.
4. If it is an operand, push its value onto the stack.
5. Otherwise, do steps 6–9:
6. Pop top operand '*a*' from the stack.
7. Pop second top operand '*b*' from the stack.
8. Evaluate given operation of operands 'a' and 'b' as c= a t b.
9. Push result '*c*' onto the stack.
10. Return the top element on the stack.

Example: Shows the evaluation of the expression [A B-C*B D+*] using 5 for A, 2 for B, 3 for C and 1 for D.



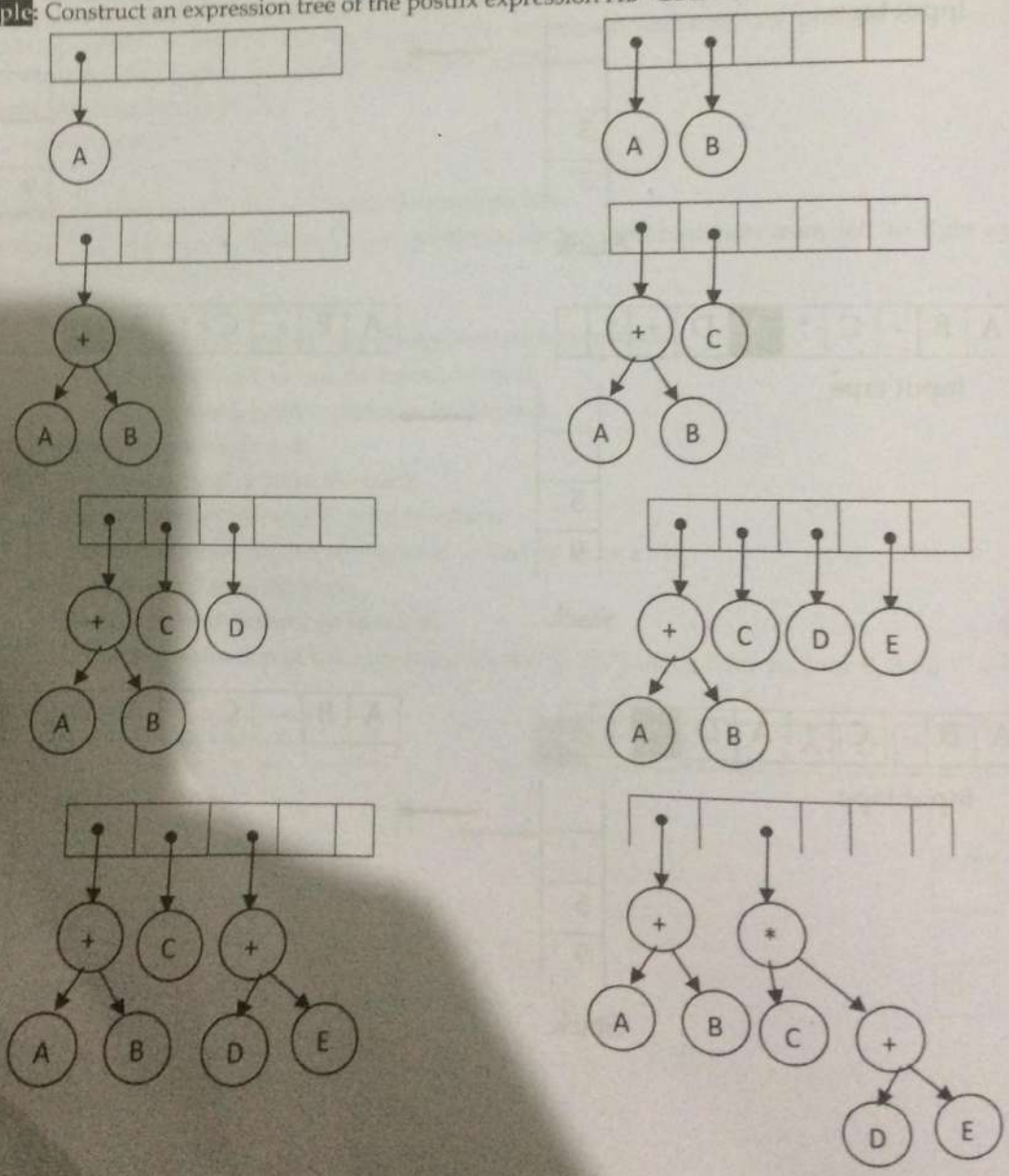

A B - C * A D + *
Input tape
2
5
Stack
A B - C * A D + *
3
A B - C * A D + *
Input tape
3
3
Stack
A B - C * A D + *
9
A B - C * A D + *
Input tape
5
9
Stack
A B - C * A D + *
1
5
9
A B - C * A D + *
Input tape
6
9
Stack
A B - C * A D + *
54

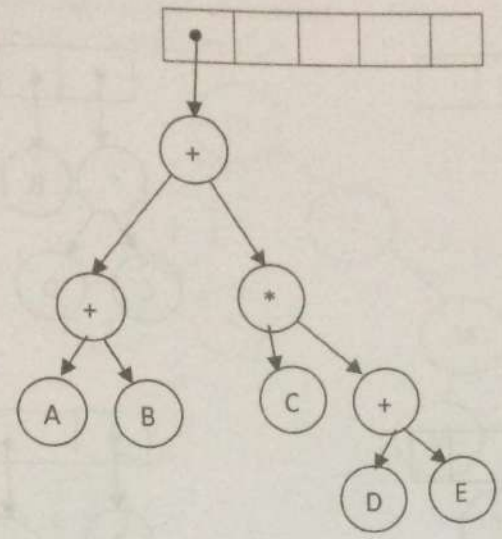
Constructing an expression tree from postfix expression

To construct an expression tree from given postfix expression follow following steps:

1. Scan given postfix expression from left to right
2. If scanned symbol is operand, then
   a. Make a tree with only one node and push its pointer to the top of the stack
3. If scanned symbol is operator, then
   a. Pop top element and place it to T2
   b. Again pop top of the element and place it to T1
4. Make a new tree with root node as given operator and its left and right sub-tree as T1 and T2 respectively
5. Continue this process until scanning pointer do not reach at last location of given postfix expression

Example: Construct an expression tree of the postfix expression AB+CDE+*$



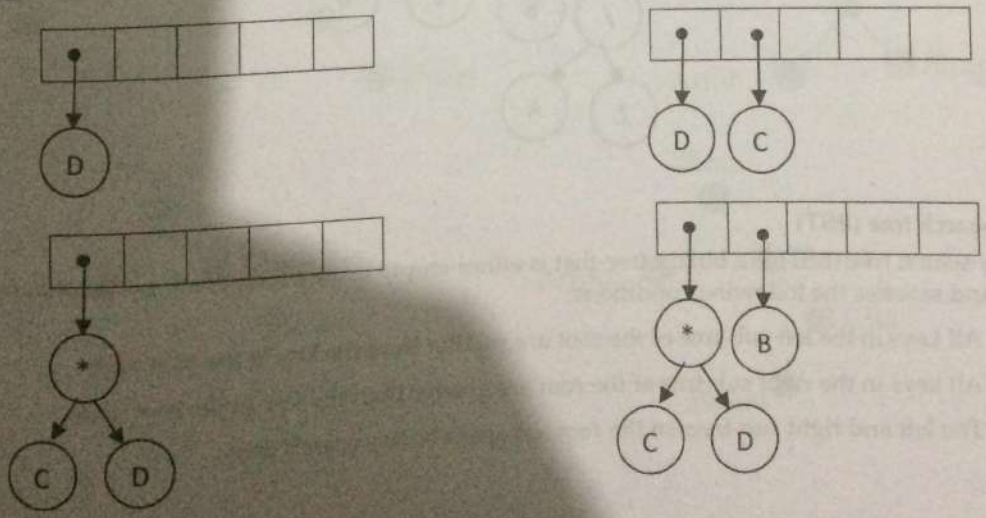


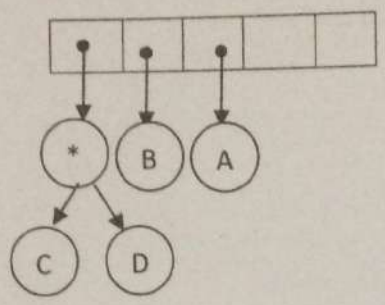
Constructing an expression tree from Prefix expression

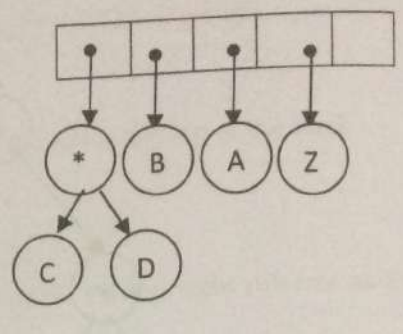
To construct an expression tree from given postfix expression follow following steps:

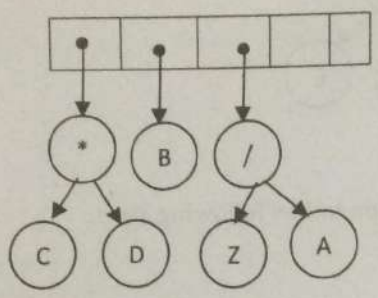
1. Start
2. Scan given postfix expression from right to left
3. If scanned symbol is operand, then
   a. Make a tree with only one node and push its pointer to the top of the stack
4. If scanned symbol is operator, then
   a. Pop top element and place it to T_1
   b. Again pop top of the element and place it to T_2
5. Make a new tree with root node as given operator and its left and right sub-tree as T1 and T2 respectively
6. Continue this process until scanning pointer do not reach at last location of given postfix expression
7. Stop

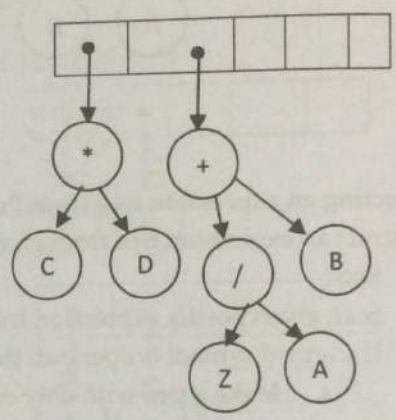
Example: Construct an expression tree of the prefix expression $+/ZAB*CD

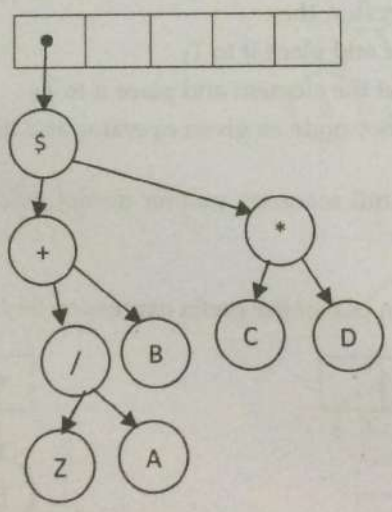












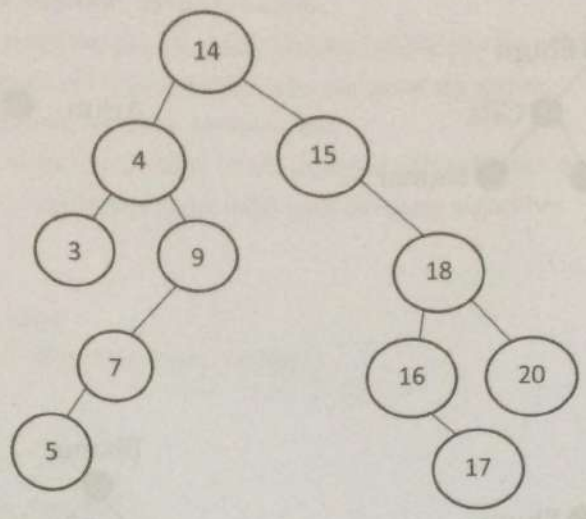
Binary search tree (BST)

A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:

- All keys in the left sub-tree of the root are smaller than the key in the root node
- All keys in the right sub-tree of the root are greater than the key in the root node
- The left and right sub-trees of the root are again binary search trees

Example 1: Given the following sequence of numbers

14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9



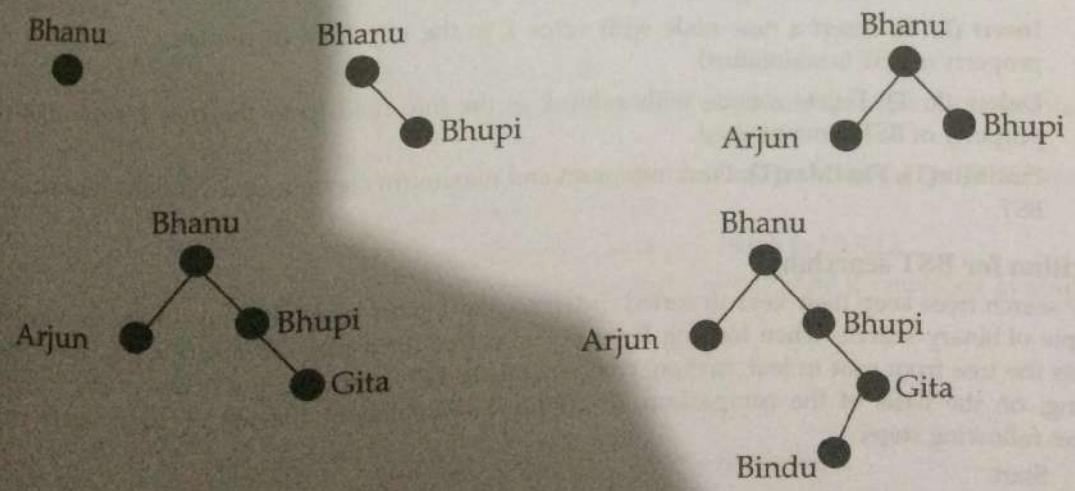
C representation for Binary search tree

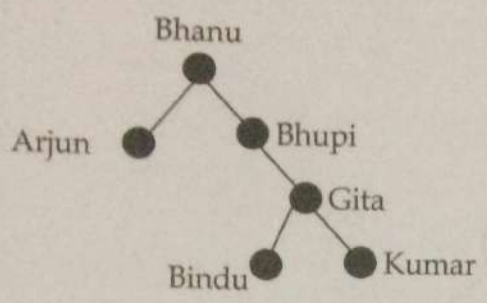
The structure of BST is described as below:

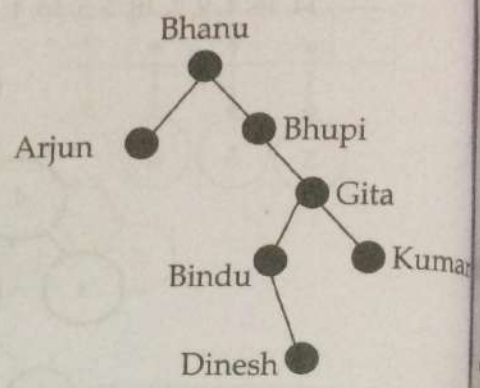
```
struct BSTNode
{
        int info;
        struct BSTNode *left;
        struct BSTNode *right;
};
struct BSTNode *root=NULL;
```

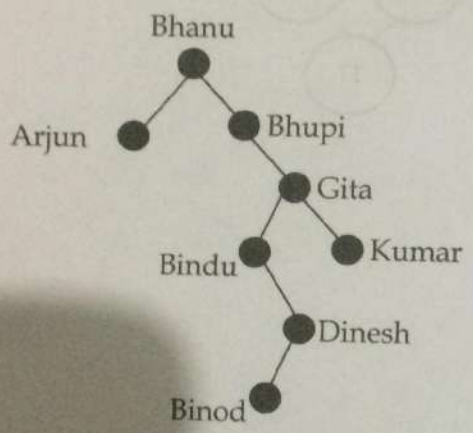
Example 2: Construct a binary search tree for the words Bhanu, Bhupi, Arjun, Gita, Bindu, Kumar, Dinesh, Binod, Umesh (using alphabetical order).

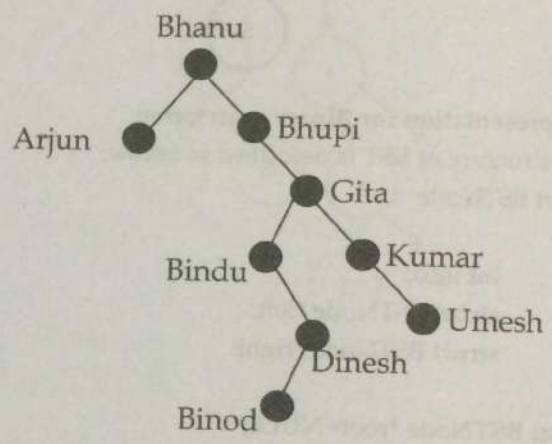
Solution:











Operations on Binary search tree (BST)

Following operations can be done in BST:

- **Search (k, T):** Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- **Insert (k, T):** Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
- **Delete (k, T):** Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- **FindMin(T), FindMax(T):** Find minimum and maximum element from the given nonempty BST.

Algorithm for BST searching

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. It involves following steps:

1. Start

2. Check, whether value in current node and searched value are equal. If so, value is found. Otherwise,
3. if searched value is less, than the node's value:
   - if current node has no left child, searched value doesn't exist in the BST;
   - Otherwise, handle the left child with the same algorithm.
4. if a new value is greater, than the node's value:
   - if current node has no right child, searched value doesn't exist in the BST;
   - Otherwise, handle the right child with the same algorithm.
5. Stop

C function for BST searching

```
BSTNode * BinSearch(struct BSTNode *root , int key)
{
    if(root == NULL)
        return NULL;
    else if (key == root→info)
        return root;
    else if(key < root→info)
        return BinSearch(root→left, key);
    else
        return BinSearch(root→right, key);
}
```

Insertion of a node in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this a search is carried out. If the search is unsuccessful, then item is inserted. The insertion operation is performed as follows:

1. Start
2. Create a newNode with given value and set its **left** and **right** to **NULL**.
3. Check whether tree is Empty or not
4. If the tree is Empty, then set root to newNode.
5. If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).
6. If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.
7. Repeat the above step until we reach to a leaf node (e.i., reach to NULL).
8. After reaching a leaf node, then isert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.
9. Stop

BST insertion function

Following is a recursive function for insertion of new nodes to existing BST.

```
void insert(struct BSTNode *root, int item)
```

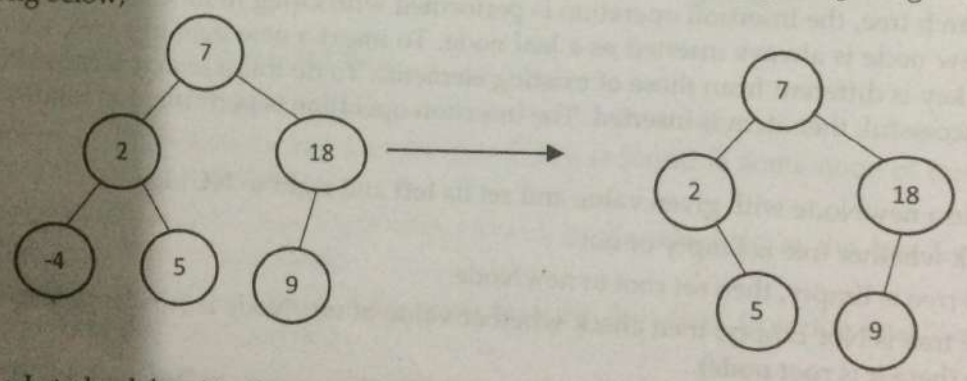
```
{
        if(root=NULL)
        {
                root=(struct BSTNode*)malloc (sizeof(struct BSTNode));
                root→left=root→right=NULL;
                root→info=item;
        }
        else
        {
                if(item<root→info)
                        root→left=insert(root→left, item);
                else
                        root→right=insert(root→right, item);
        }
}
```

Deleting a node from the BST

While deleting a node from BST, first we need to perform a search for deleting element. Once we have found the node to be deleted, there may be three cases shown in given below. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two sub-trees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has.

1. The node to be deleted may be a leaf node

The node is a leaf; it has no children. This is the easiest case. The appropriate reference of its parent is set to null and the space occupied by the deleted node is later claimed by the garbage collector as shown in fig below;

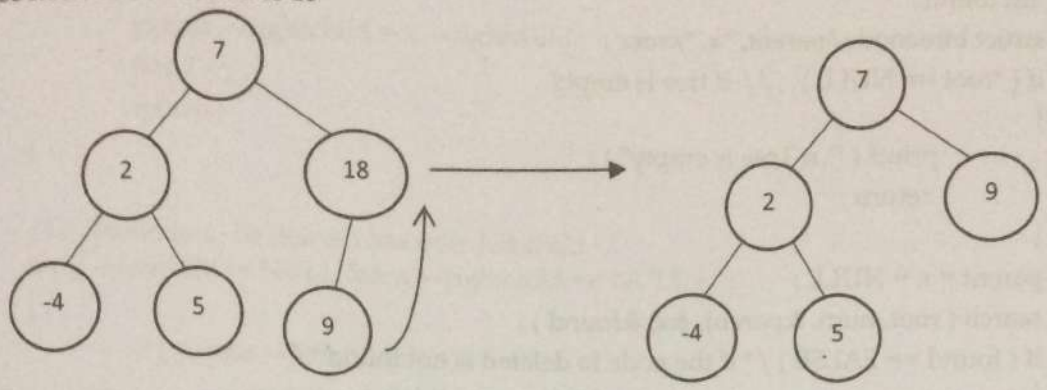


Suppose node to be deleted is -4

2. The node to be deleted has one child

In this case the child of the node to be deleted is appended to its parent node. In this way, the node's children are lifted up by one level as shown in fig below:

Suppose node to be deleted is 18

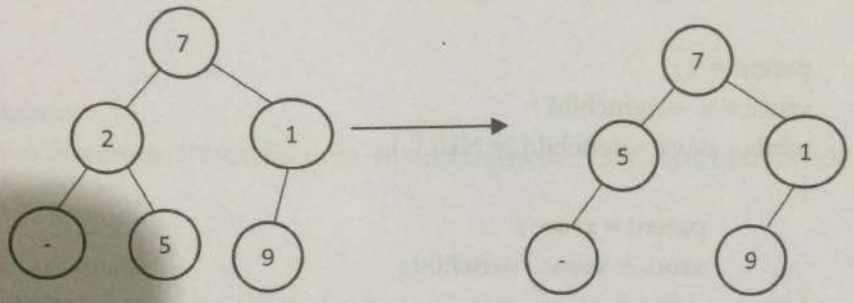


3. The node to be deleted has two children

In this case node to be deleted is replaced by its in-order successor node.

OR

The node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.



Suppose node to deleted is 2

Find minimum element in the right sub-tree of the node to be removed. In current example it is 5.

General algorithm to delete a node from a BST

1. Start
2. If a node to be deleted is a leaf node at left side, then simply delete and set null pointer to its parent's left pointer.
3. If a node to be deleted is a leaf node at right side, then simply delete and set null pointer to its parent's right pointer
4. If a node to be deleted has on child, then connect its child pointer with its parent pointer and delete it from the tree
5. If a node to be deleted has two children, then replace the node being deleted either by a right most node of its left sub-tree or left most node of its right sub-tree (replace their in-order successor node).
6. End

Function in C for deleting element from the BST

Here after deleting node from BST, we have two sub-trees. Thus we need to merge these two nodes properly. This type of deleting element from BST is called deletion by merging operation.

```
void delete ( struct btreenode **root, int num )
{
```

```
int found ;
struct btreenode *parent, *x, *xsucc ;
if ( *root == NULL )   // if tree is empty
{
        printf ( "\n Tree is empty" ) ;
        return ;
}
parent = x = NULL ;
search ( root, num, &parent, &x, &found ) ;
if ( found == FALSE ) /* if the node to deleted is not found */
{
        printf ( "\n Data to be deleted, not found" ) ;
        return ;
}
/* if the node to be deleted has two children */
if ( x →leftchild != NULL && x →rightchild != NULL )
{
        parent = x ;
        xsucc = x →rightchild ;
        while ( xsucc →leftchild != NULL )
        {
                parent = xsucc ;
                xsucc = xsucc →leftchild ;
        }
        x →data = xsucc →data ;
        x = xsucc ;
}

/* if the node to be deleted has no child */
if ( x →leftchild == NULL && x →rightchild == NULL )
{
        if ( parent →rightchild == x )
                parent →rightchild = NULL ;
        else
                parent →leftchild = NULL ;

        free ( x ) ;
        return ;
}
/* if the node to be deleted has only rightchild */
if ( x →leftchild == NULL && x →rightchild != NULL )
{
        if ( parent →leftchild == x )
        parent →leftchild = x →rightchild ;
```

```
            else
            parent →rightchild = x →rightchild ;
            free ( x ) ;
            return ;
        }

        /* if the node to be deleted has only left child */
        if ( x →leftchild != NULL && x →rightchild == NULL )
        {
            if ( parent →leftchild == x )
                    parent →leftchild = x →leftchild ;
            else
            parent →rightchild = x →leftchild ;
            free ( x ) ;
            return ;
        }
}
```

BST search function

```
void search ( struct btreenode **root, int num, struct btreenode **par, struct btreenode **x, int
            *found )
{
        Struct btreenode *q ;
        q = *root ;
        *found = FALSE;
        *par = NULL ;
        while ( q != NULL )
        {
                if ( q →data == num )
                {
                *found = TRUE ;
                *x = q ;
                return ;
                }
                *par = q ;
                if ( q →data > num )
                q = q →leftchild ;
                else
                q = q →rightchild ;
        }
}
```

Complete menu driven program for implementing BST in C

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

```
# include <stdlib.h>
typedef struct BST
{
        int data;
        struct BST *lchild, *rchild;
} node;
void insert(node *, node *);
void inorder(node *);
void preorder(node *);
void postorder(node *);
node *search(node *, int, node **);
void main()
{
        int choice;
        char ans = 'N';
        int key;
        node *new_node, *root, *tmp, *parent;
        node *get_node();
        root = NULL;
        clrscr();
        printf("\nProgram For Binary Search Tree ");
        do
        {
                printf("\n1.Create");
                printf("\n2.Search");
                printf("\n3.Recursive Traversals");
                printf("\n4.Exit");
                printf("\nEnter your choice :");
                scanf("%d", &choice);
                switch (choice)
                {
                    case 1:
                        do {
                        new_node = get_node();
                        printf("\nEnter The Element ");
                        scanf("%d", &new_node→data);
                        if (root == NULL)     /* Tree is not Created */
                                root = new_node;
                        else
                                insert(root, new_node);
                        printf("\n Want To enter More Elements?(y/n)");
                        ans = getch();
                        } while (ans == 'y');
```

```
                break;
        case 2:
                printf("\n Enter Element to be searched :");
                scanf("%d", &key);
                tmp = search(root, key, &parent);
                printf("\n Parent of node %d is %d", tmp→data, parent→data);
                break;
        case 3:
                if (root == NULL)
                printf("Tree Is Not Created");
                else
                {
                printf("\n The Inorder display : ");
                inorder(root);
                printf("\n The Preorder display : ");
                preorder(root);
                printf("\n The Postorder display : ");
                postorder(root);
                }
                break;
        }
 } while (choice != 4);
}

node *get_node()//Get new Node
{
        node *temp;
        temp = (node *) malloc(sizeof(node));
        temp→lchild = NULL;
        temp→rchild = NULL;
        return temp;
}

/* This function is for creating a binary search tree */
void insert(node *root, node *new_node)
{
        if (new_node→data < root→data)
        {
                if (root→lchild == NULL)
                        root→lchild = new_node;
                else
                        insert(root→lchild, new_node);
        }
```

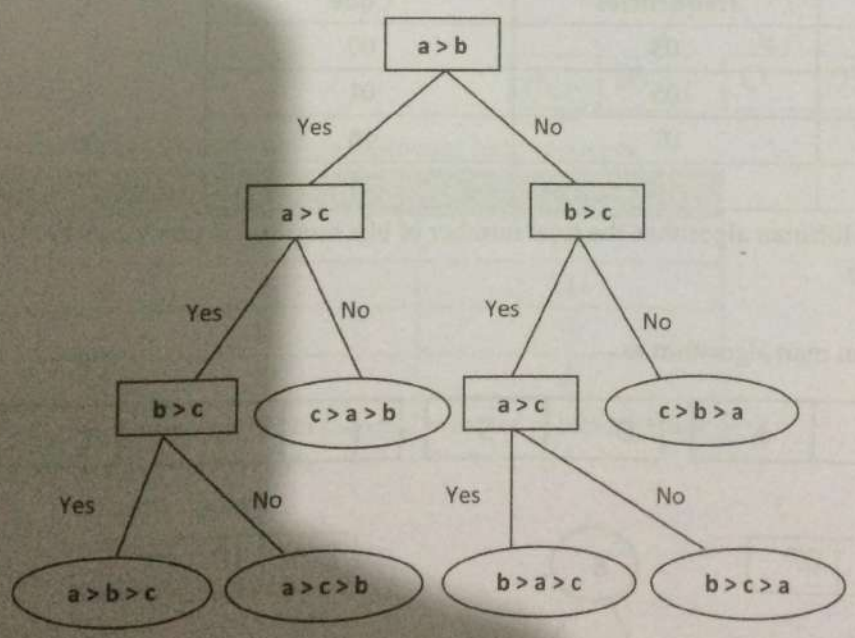
```
        if (new_node→data > root→data)
        {
                if (root→rchild == NULL)
                        root→rchild = new_node;
                else
                        insert(root→rchild, new_node);
        }
}

node *search(node *root, int key, node **parent)
{
        node *temp;
        temp = root;
        while (temp != NULL)
        {
                if (temp→data == key)
                {
                        printf("\n The %d Element is Present", temp→data);
                        return temp;
                }
                *parent = temp;
                if (temp→data > key)
                        temp = temp→lchild;
                else
                        temp = temp→rchild;
        }
        return NULL;
}

void inorder(node *temp)
{
        if (temp != NULL)
        {
                inorder(temp→lchild);
                printf("%d", temp→data);
                inorder(temp→rchild);
        }
}
void preorder(node *temp)
{
        if (temp != NULL)
        {
                printf("%d", temp→data);
```

```
            preorder(temp→lchild);
            preorder(temp→rchild);
        }
}

void postorder(node *temp)
{
        if (temp != NULL)
        {
            postorder(temp→lchild);
            postorder(temp→rchild);
            printf("%d", temp→data);
        }
}
```

Decision trees

Binary trees can be used to locate items based on comparisons, in this situation each comparison tells us to visit either left sub-tree or right sub-tree. A rooted tree where internal vertices correspond to a decision and sub-tree at these vertices gives possible outcomes of the decisions made is called decision tree.

Example: A decision tree that orders the elements of the list a, b, c.



Huffman algorithm

"Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights."

This method is mainly applicable to many forms of data transmission. In 1951, David Huffman found the "most efficient method of representing numbers, letters, and other symbols using binary code". Now standard method used for data compression. In Huffman Algorithm, a set of nodes assigned with values if fed to the algorithm.

Initially 2 nodes are considered and their sum forms their parent node. When a new element is considered, it can be added to the tree. Its value and the previously calculated sum of the tree are used to form the new node which in turn becomes their parent. Let us take any four characters and their frequencies, and sort this list by increasing frequency.

Since to represent 4 characters the 2 bit is sufficient thus take initially two bits for each character this is called fixed length character.

Character	frequencies
A	03
T	07
E	10
O	05

Now sort these characters according to their frequencies in non-decreasing order.

Character	frequencies	Code
A	03	00
O	05	01
T	07	10
E	10	11

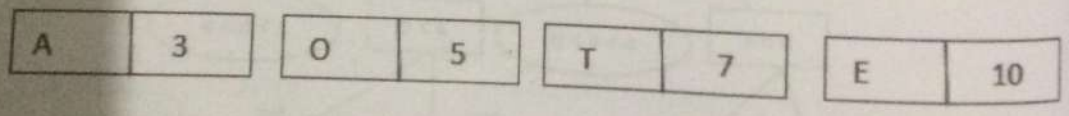
Here before using Huffman algorithm the total number of bits required is nb=3*2+5*2+7*2+10*2

=06+10+14+20

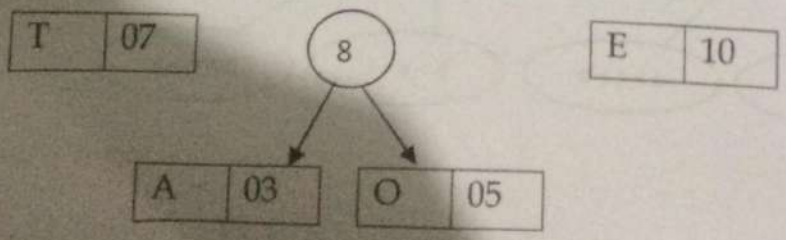
=50bits

Now using Huffman man algorithm as,

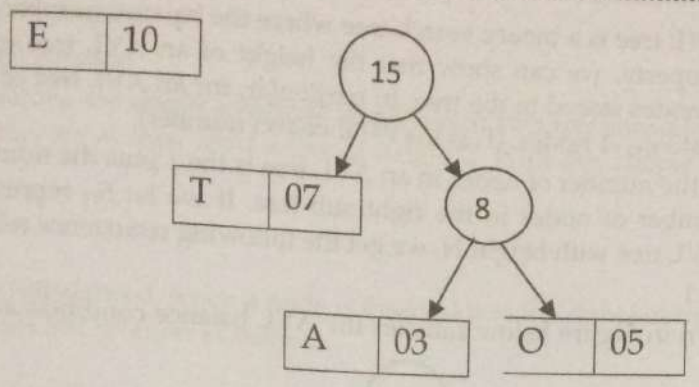
Step 1:



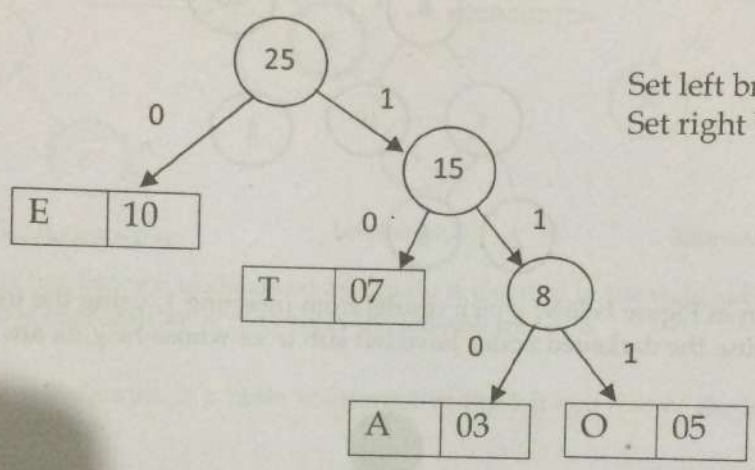
Step 2:



Step 3:



Step 4:



Set left branch=0 &
Set right branch=1

Now from variable length code we get following code sequence.

Character	frequencies	Code
A	03	110
O	05	111
T	07	10
E	10	0

Thus after using Huffman algorithm the total number of bits required is:

Number of bits = 3*3+5*3+7*2+10*1
= 09+15+14+10
=48 bits
= (50-48)/50*100%
= 4%

Since in this small example, we can save about 4% of space by using Huffman algorithm. If we take large example with a lot of characters and their frequencies, we can save a lot of space.

AVL tree

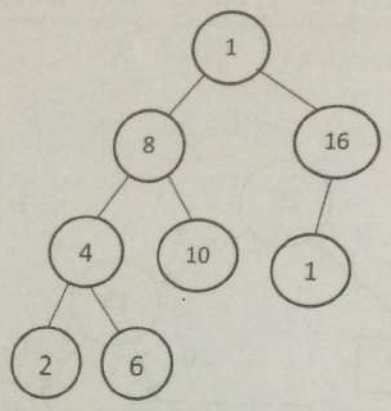
The first balanced binary tree is the AVL tree, named after their inventor **Adelson, Velski** and **Landis, AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called

the **Balance Factor**. An AVL tree is a binary search tree where the balance number at each node is -1, 0, or 1. Based on this property, we can show that the height of an AVL tree is logarithmic with respect to the number of nodes stored in the tree. In particular, for an AVL tree of height H, we find that it must contain at least F_{H+3} -1 nodes. (F_i is the ith Fibonacci number).

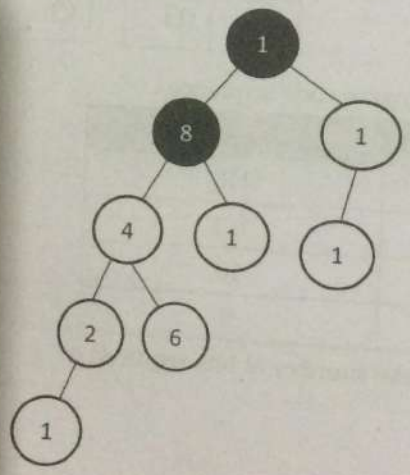
To prove this, notice that the number of nodes in an AVL tree is the 1 plus the number of notes in the left sub-tree plus the number of nodes in the right sub-tree. If we let S_H represent the minimum number of nodes in an AVL tree with height H, we get the following recurrence relation:

$$S_H = S_{H-1} + S_{H-2} + 1$$

Example: The tree shown in Figure below satisfies the AVL balance condition and is thus an AVL tree.



The tree shown in Figure below, which results from inserting 1, using the usual algorithm, is not an AVL tree because the darkened nodes have left sub trees whose heights are 2 larger than their right sub trees.



How to make non AVL tree to AVL tree?

We now that,

Balance Factor = height (left subtree) – height (right subtree)

If the difference in the height of left and right sub-trees is more than 1, or less than 0 then the tree is unbalanced and we need to make height balance by using some rotation techniques. To balance itself, an AVL tree may perform the following four kinds of rotations:

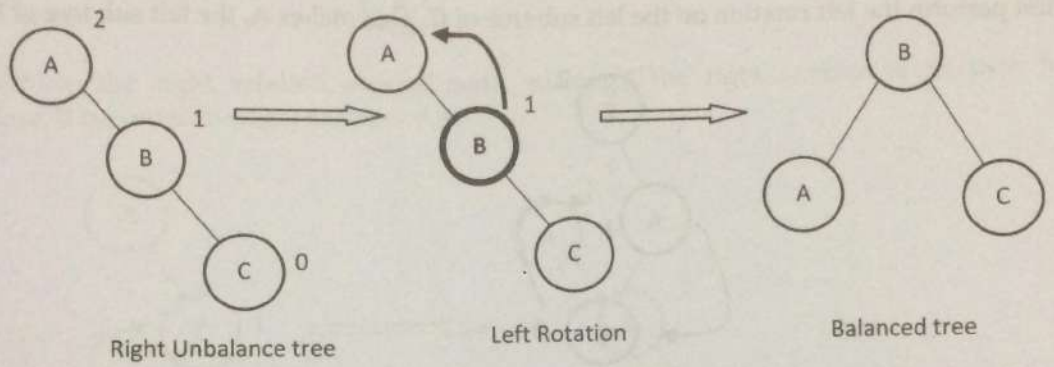
- *Left rotation*
- *Right rotation*

- Left-Right rotation
- Right-Left rotation

The first two rotations are **single rotations** and the next two rotations are **double rotations**. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

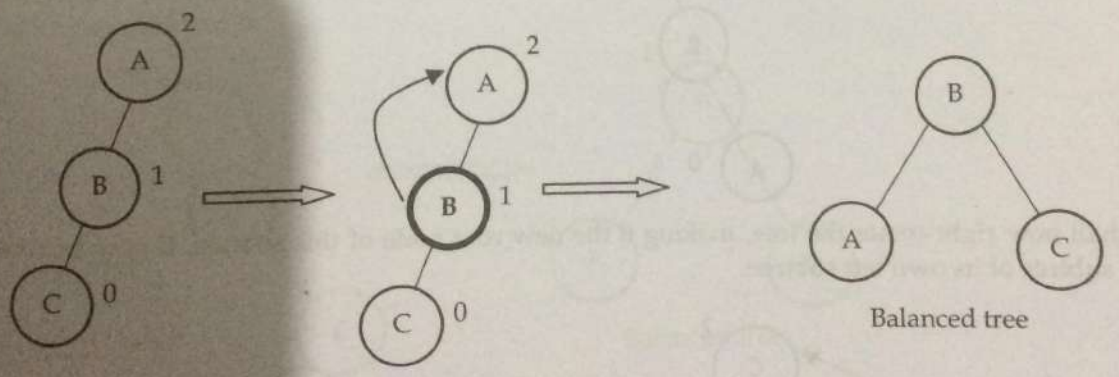
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation as below,



In our example, node **A** has become unbalanced as a node is inserted in the right sub tree of A's right sub tree. We perform the left rotation by making **A** the left-sub tree of B.

Right 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.

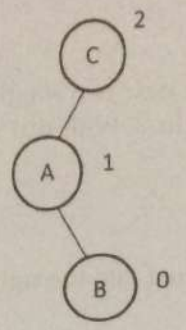


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

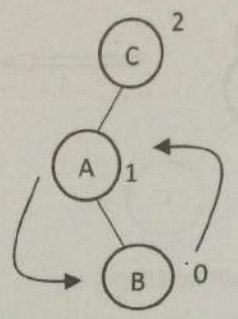
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

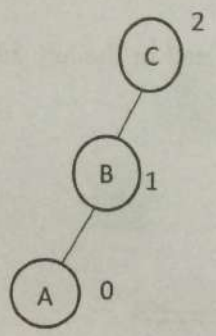
A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.



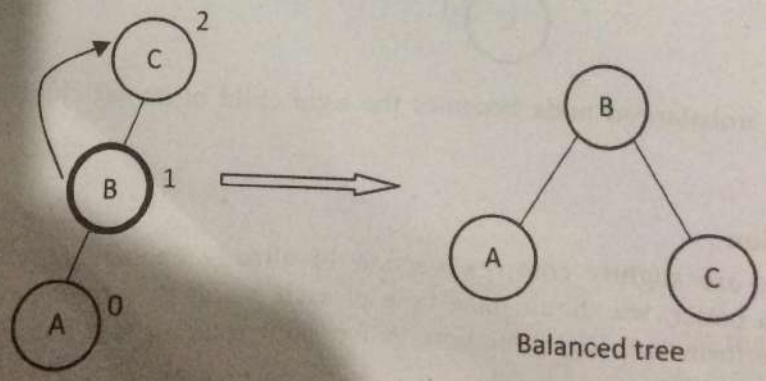
We first perform the left rotation on the left sub tree of **C**. This makes **A**, the left sub tree of **B**.



Node **C** is still unbalanced; however now, it is because of the left-sub tree of the left-sub tree.



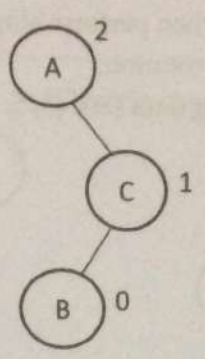
We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.



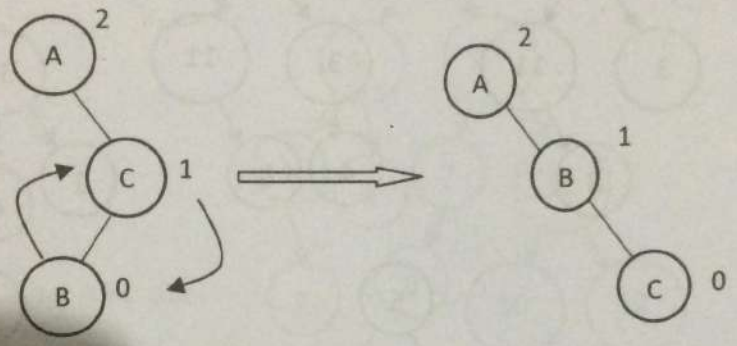
Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

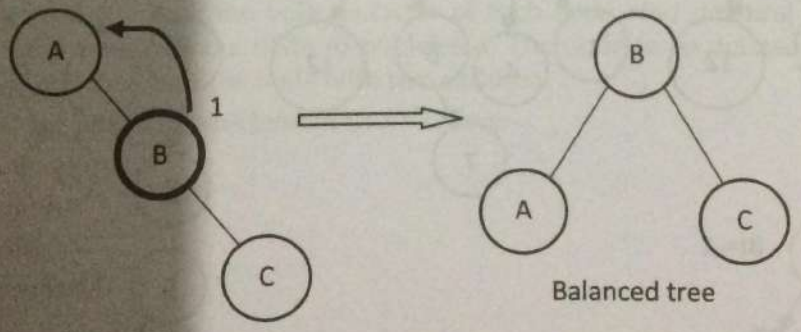
A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.



First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation. A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.



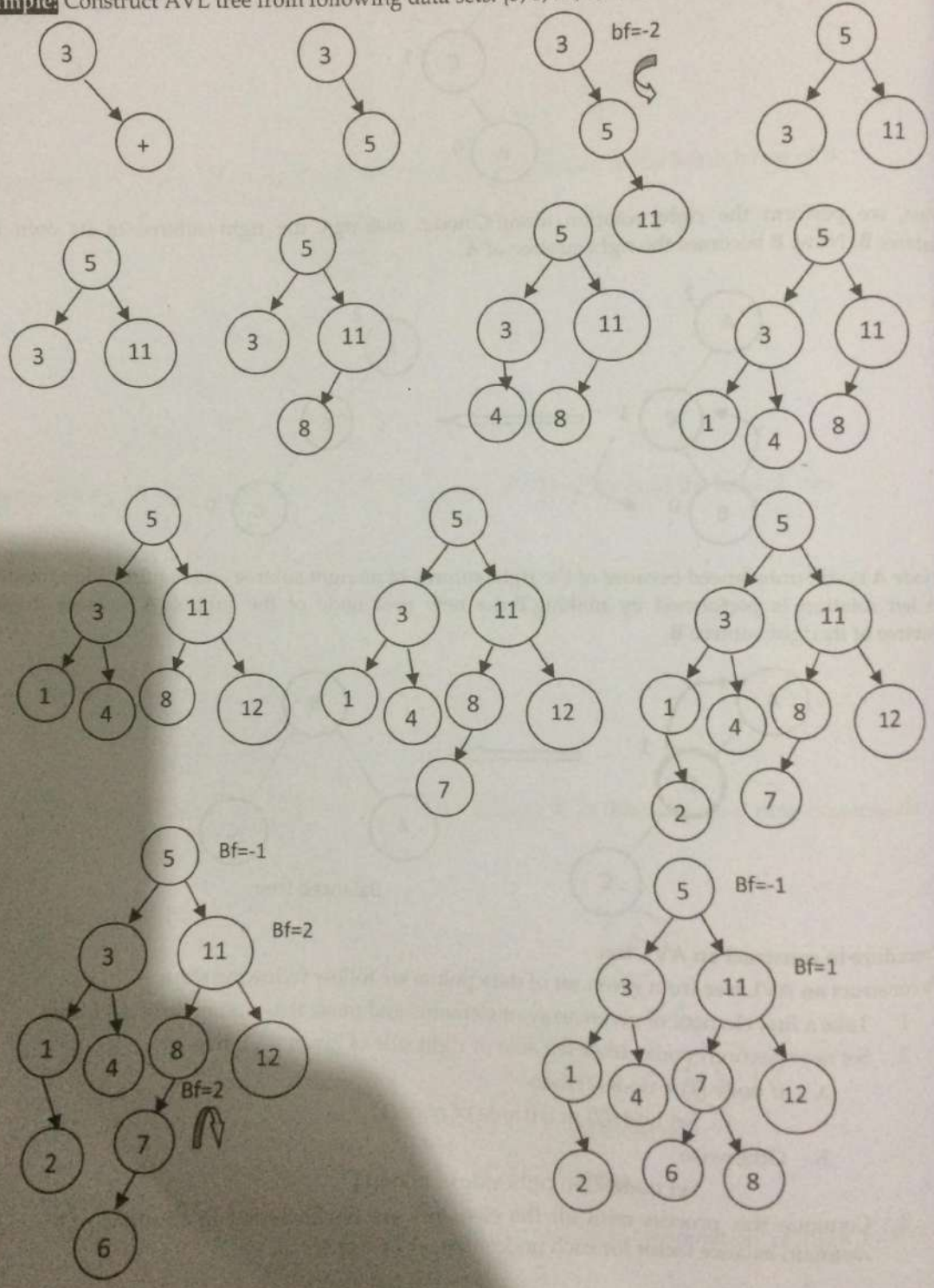
Procedure to construct an AVL tree

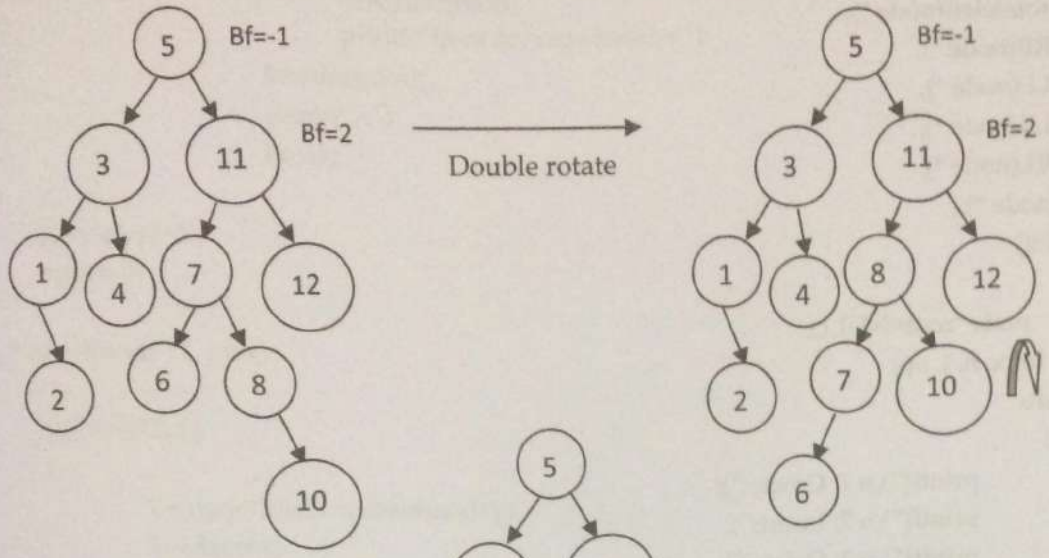
To construct an AVL tree from given set of data points we follow following steps:

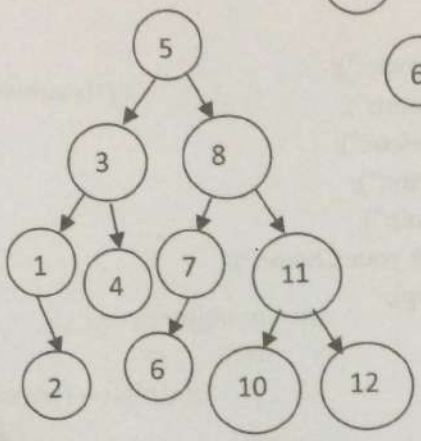
1. Take a first element of given array of elements and make it as first node of AVL tree
2. Set next (second) node either left side or right side of given AVL tree
   a. If node (1) > node(2) then
      i. Set node(2) in left side of node(1)
   b. Otherwise
      i. Set node(2) in right side of node(1)
3. Continue this process until all the elements are not included in resulting AVL tree and maintain balance factor for each node either -1 or 0 or 1.

4. If balance factor of a particular node is not in given range (-1 to 1) then rotate either single or double
   a. If straight path take place, then perform single rotation
   b. Otherwise perform double rotation

Example: Construct AVL tree from following data sets: {3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10}







This is required final AVL tree

Deleting data element from AVL tree

Deleting element from AVL tree is same as that of deleting element from BST. Slightly different is that here we need to check the balance factor of each node after deleting the node from the tree. Initially we need to search the node to be deleted. The node to be deleted could be a leaf node, a node with one child node of the node with two children.

Program in C for inserting an element in AVL Tree

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct node
{
        int data;
        struct node *left,*right;
        int ht;
}node;
node *insert(node *,int);
node *Delete(node *,int);
void preorder(node *);
void inorder(node *);
int height( node *);
node *rotateright(node *);
```

```
node *rotateleft(node *);
node *RR(node *);
node *LL(node *);
node *LR(node *);
node *RL(node *);
int BF(node *);
int main()
{
        node *root=NULL;
       int x, n, i, op;
       do
       {
            printf("\n 1: Create:");
            printf("\n 2: Insert:");
            printf("\n 3: Delete:");
            printf("\n 4: Print:");
            printf("\n 5: Quit:");
            printf("\n Enter your Choice:");
            scanf("%d", &op);
            switch(op)
            {
                  case 1:
                              printf("Enter no. of elements:");
                        scanf("%d", &n);
                        printf("Enter tree data:");
                        root=NULL;
                        for(i=0;i<n;i++)
                        {
                              scanf("%d", &x);
                              root=insert(root, x);
                        }
                        break;
                  case 2:
                              printf("Enter a data:");
                              scanf("%d",&x);
                              break;                                    root=insert(root, x);
                     case 3:
                              printf("Enter a data:");
                              scanf("%d",&x);
                              root=Delete(root, x);
                              break;
                     case 4:
                              printf("Preorder sequence:\n");
```

```
                    preorder(root);
                    printf("In-order sequence:\n");
                inorder(root);
                printf("n");
                break;
        }
    }while(op!=5);
    return 0;
}
node * insert(node *T, int x)
{
    if(T==NULL)
    {
        T=(node*)malloc(sizeof(node));
        T→data=x;
        T→left=NULL;
        T→right=NULL;
    }
    else
        if(x > T→data)          // insertin right sub-tree
        {
            T→right=insert(T→right, x);
            if(BF(T)==-2)
                if(x>T→right→data)
                    T=RR(T);
                else
                    T=RL(T);
        }
        else
            if(x < T→data)
            {
                T→left=insert(T→left, x);
                if(BF(T)==2)
                    if(x < T→left→data)
                        T=LL(T);
                    else
                        T=LR(T);
            }
            T→ht=height(T);
            return(T);
}
node * Delete(node *T, int x)
{
```

```
node *p;
if(T==NULL)
{
        return NULL;
}
else
        if(x > T→data)          // insert in right sub-tree
        {
                T→right=Delete(T→right, x);
                if(BF(T)==2)
                        if(BF(T→left)>=0)
                                T=LL(T);
                        else
                                T=LR(T);
        }
        else
                if(x<T→data)
                {
                        T→left=Delete(T→left, x);
                        if(BF(T)==-2)  //Rebalance during windup
                                if(BF(T→right)<=0)
                                        T=RR(T);
                                else
                                        T=RL(T);
                }
                else
                {
                  //data to be deleted is found
                  if(T→right!=NULL)
                    {
                            p=T→right;
                            while(p→left!= NULL)
                                    p=p→left;
                            T→data=p→data;
                                    T→right=Delete(T→right, p→data);
                            if(BF(T)==2)//Rebalance during windup
                                    if(BF(T→left)>=0)
                                            T=LL(T);
                                    else
                                            T=LR(T);
                    }
                  else
                        return(T→left);
```

```
            }
        T→ht=height(T);
        return(T);
}

int height(node *T)
{
    int lh, rh;
    if(T==NULL)
        return(0);
    if(T→left==NULL)
        lh=0;
    else
        lh=1+T→left→ht;
    if(T→right==NULL)
        rh=0;
    else
        rh=1+T→right→ht;
    if(lh>rh)
        return(lh);
    return(rh);
}
node * rotateright(node *x)
{
    node *y;
    y=x→left;
    x→left=y→right;
    y→right=x;
    x→ht=height(x);
    y→ht=height(y);
    return(y);
}
node * rotateleft(node *x)
{
    node *y;
    y=x→right;
    x→right=y→left;
    y→left=x;
    x→ht=height(x);
```

```
        y→ht=height(y);
        return(y);
}
node * RR(node *T)
{
        T=rotateleft(T);
        return(T);
}
node * LL(node *T)
{
        T=rotateright(T);
        return(T);
}
node * LR(node *T)
{
        T→left=rotateleft(T→left);
        T=rotateright(T);
        return(T);
}
node * RL(node *T)
{
        T→right=rotateright(T→right);
        T=rotateleft(T);
        return(T);
}
int BF(node *T)
{
        int lh, rh;
        if(T==NULL)
        return(0);
        if(T→left==NULL)
                lh=0;
        else
                lh=1+T→left→ht;
        if(T→right==NULL)
                rh=0;
        else
                rh=1+T→right→ht;
        return(lh-rh);
```

```
}
void preorder(node *T)
{
        if(T!=NULL)
        {
                printf("%d(Bf=%d)",T→data, BF(T));
                preorder(T→left);
                preorder(T→right);
        }
}
void inorder(node *T)
{
        if(T!=NULL)
        {
                inorder(T→left);
                printf("%d(Bf=%d)", T→data, BF(T));
                inorder(T→right);
        }
}
```

Time Complexity

The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(log n). So time complexity of AVL insert is O(log n).

Red-black trees

A historically popular alternative to the AVL tree is the red-black tree, in which a single top-down pass can be used during the insertion and deletion routines. This approach contrasts with an AVL tree, in which a pass down the tree is used to establish the insertion point and a second pass up the tree is used to update heights and possibly rebalance. As a result, a careful non-recursive implementation of the red-black tree is simpler and faster than an AVL tree implementation. As on AVL trees, operations on red-black trees take logarithmic worst-case time.

A red-black tree is a binary search tree having the following ordering properties:

1. Every node is colored either red or black.
2. The root is black.
3. If a node is red, its children must be black.
4. Every path from a node to a null link must contain the same number of black nodes.

In this discussion of red-black trees, shaded nodes represent red nodes. Figure below shows a red-black tree. Every path from the root to a **null** node contains three black nodes and colored node represents the red node.

Example: Read black tree with black height three

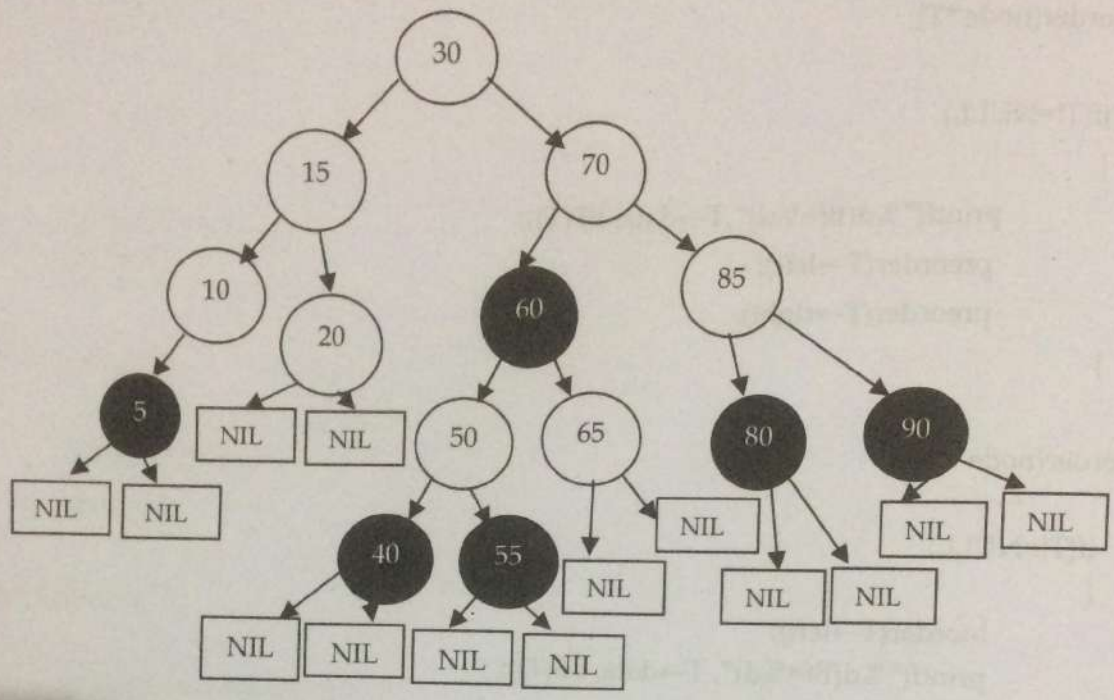


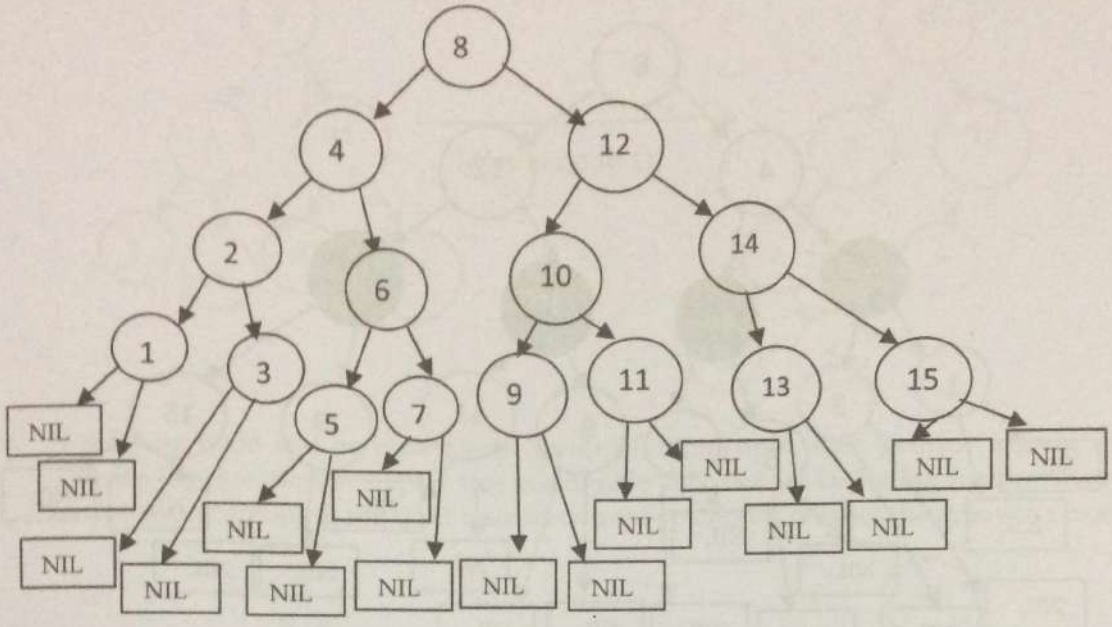
Fig: Red Black tree

Since red black tree is a BST with one extra attribute for each of the node: the color, which is either red or black. We also need to keep the track of the parent of each node. So that the red black tree has following structure;

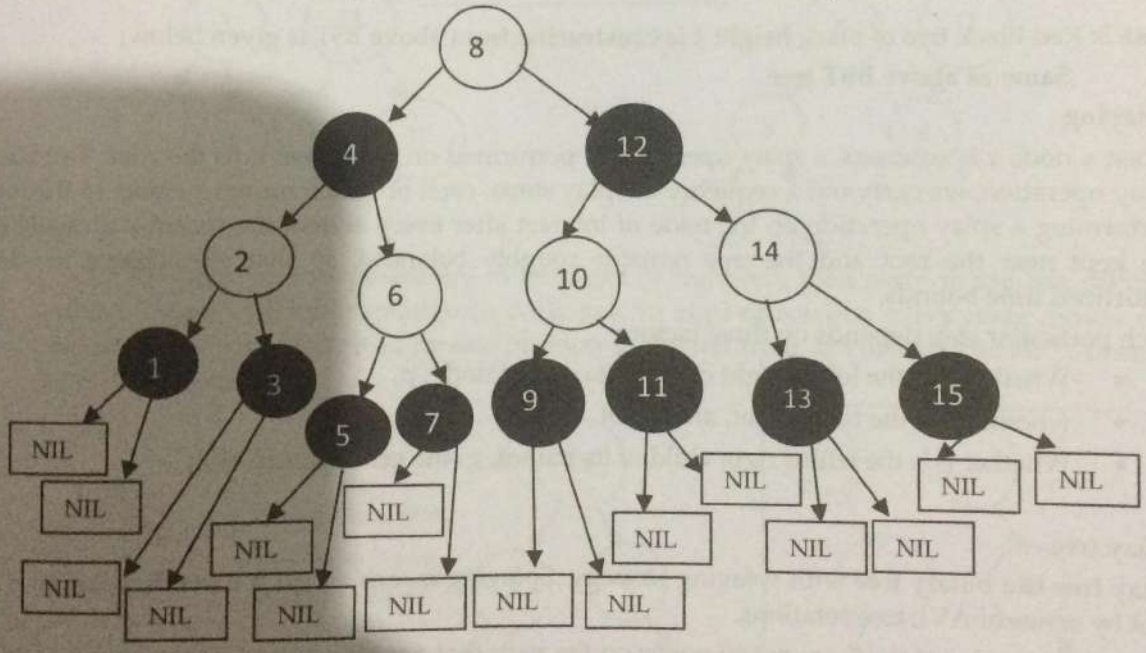
```
struct RB_Node
{
        enum {red, black} color;
        int info;
        struct RB_Node *left;
        struct RB_Node *right;
        struct RB_Node *parent;
};
struct RB_Node *root;
root=null;
```

Example 2: Draw the complete BST of length three on the keys {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15} and add the nil leaves and color the nodes in three different ways such that the black heights of the resulting tree are 2, 3 and 4.

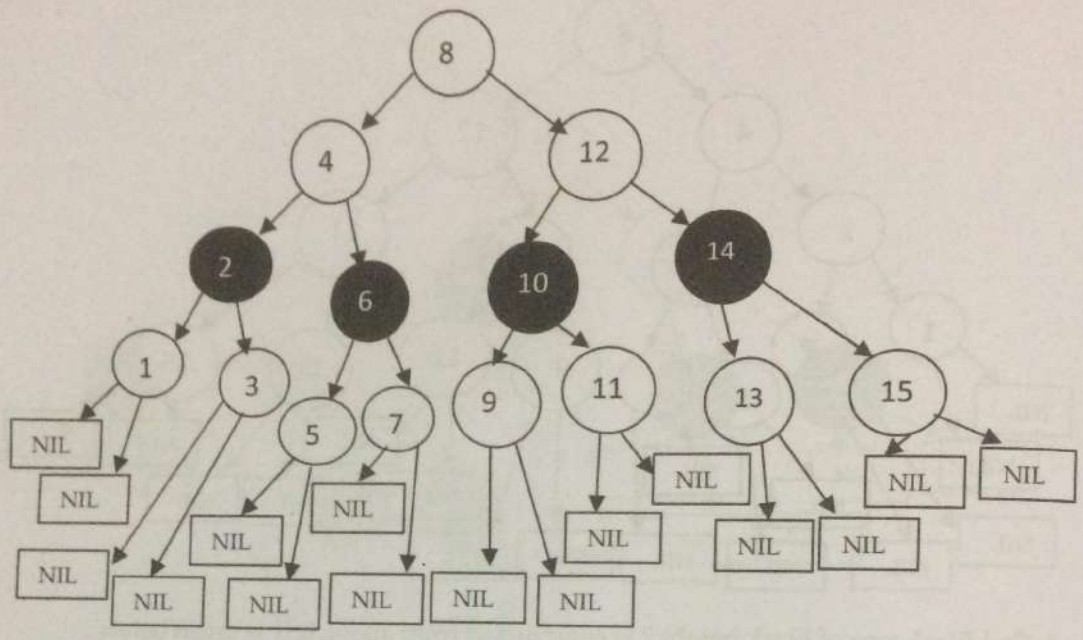
Solution: the BST of height 3 is constructed as below,



Case 1: Red Black tree of black height 2 is constructed from above BST is given below;



Case 2: Red Black tree of black height 3 is constructed from above BST is given below;



Case 3: Red Black tree of black height 4 is constructed from above BST is given below;

Same as above BST tree

Splaying

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).

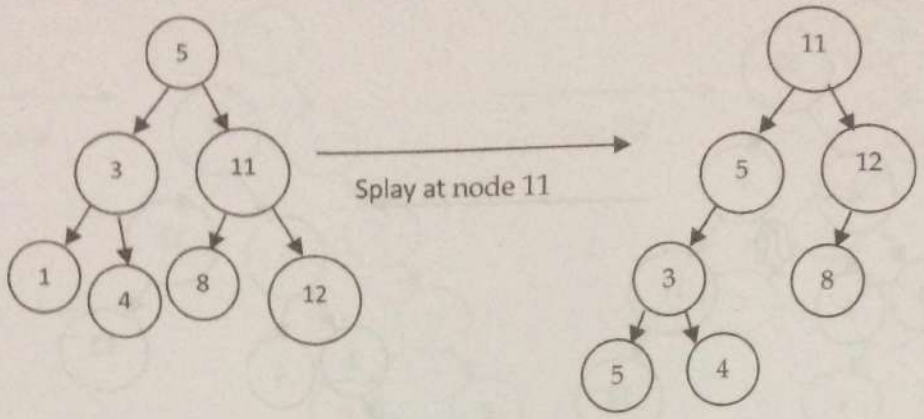
Splay tree

Splay tree is a binary tree with splaying strategy. Splaying means access a node by pushing it to the root by series of AVL tree rotations.

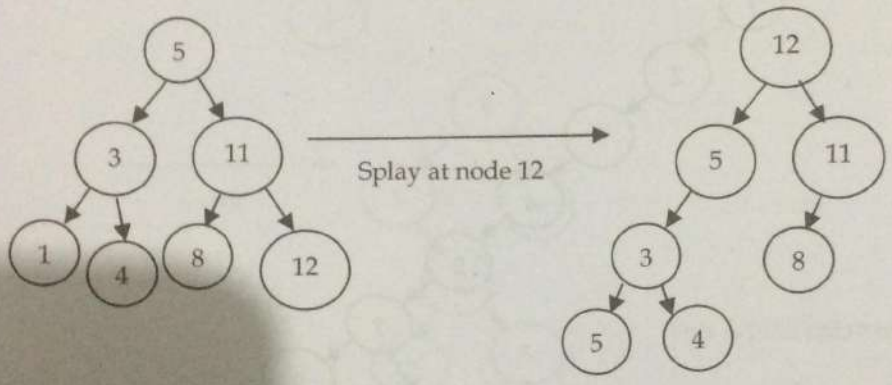
If a node is deep and there are many nodes on the path that are also relatively deep, then in this case splaying make future accesses cheaper on all these nodes. Splay tree do not require the maintenance of height or balance information, thus saving space and simplifying the coding.

Splaying strategy

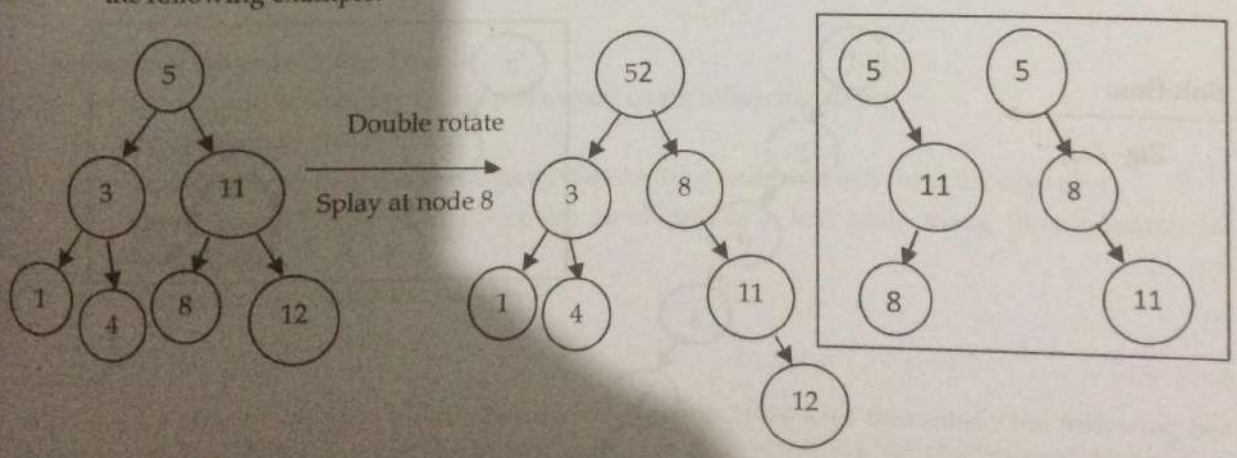
- Rotate bottom up along the access path
- If the parent of splaying node is root, then perform **zig** rotation. The **Zig Rotation** in a splay tree is similar to the single right rotation in AVL Tree rotations. In **zig** rotation every node moves one position to the right from its current position. Consider the following example:

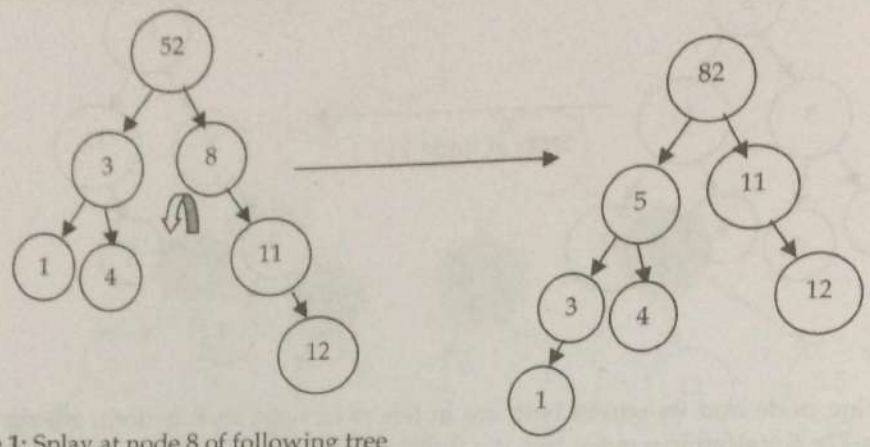


- If splaying node and its parent both are in left or in right, then perform **zig-zig** rotation. The **Zig-Zig Rotation** in a splay tree is a double zig rotation. In **zig-zig** rotation every node moves two positions to the right from its current position. Consider the following example

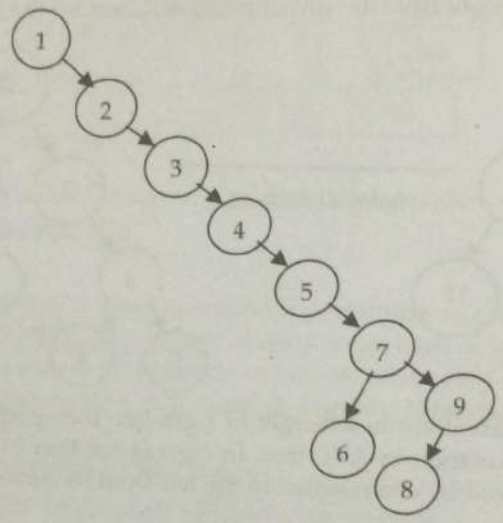


- If splaying node and its parent are in left-right or right-left, then perform **zig-zag** rotation which is same as double rotation in AVL tree. In zig-**zag** rotation every node moves one position to the right followed by one position to the left from its current position. Consider the following example:



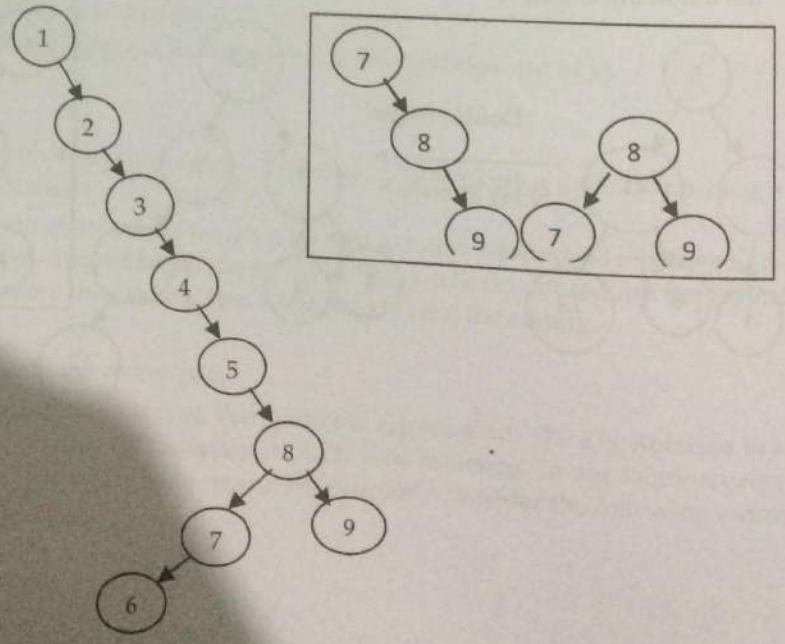


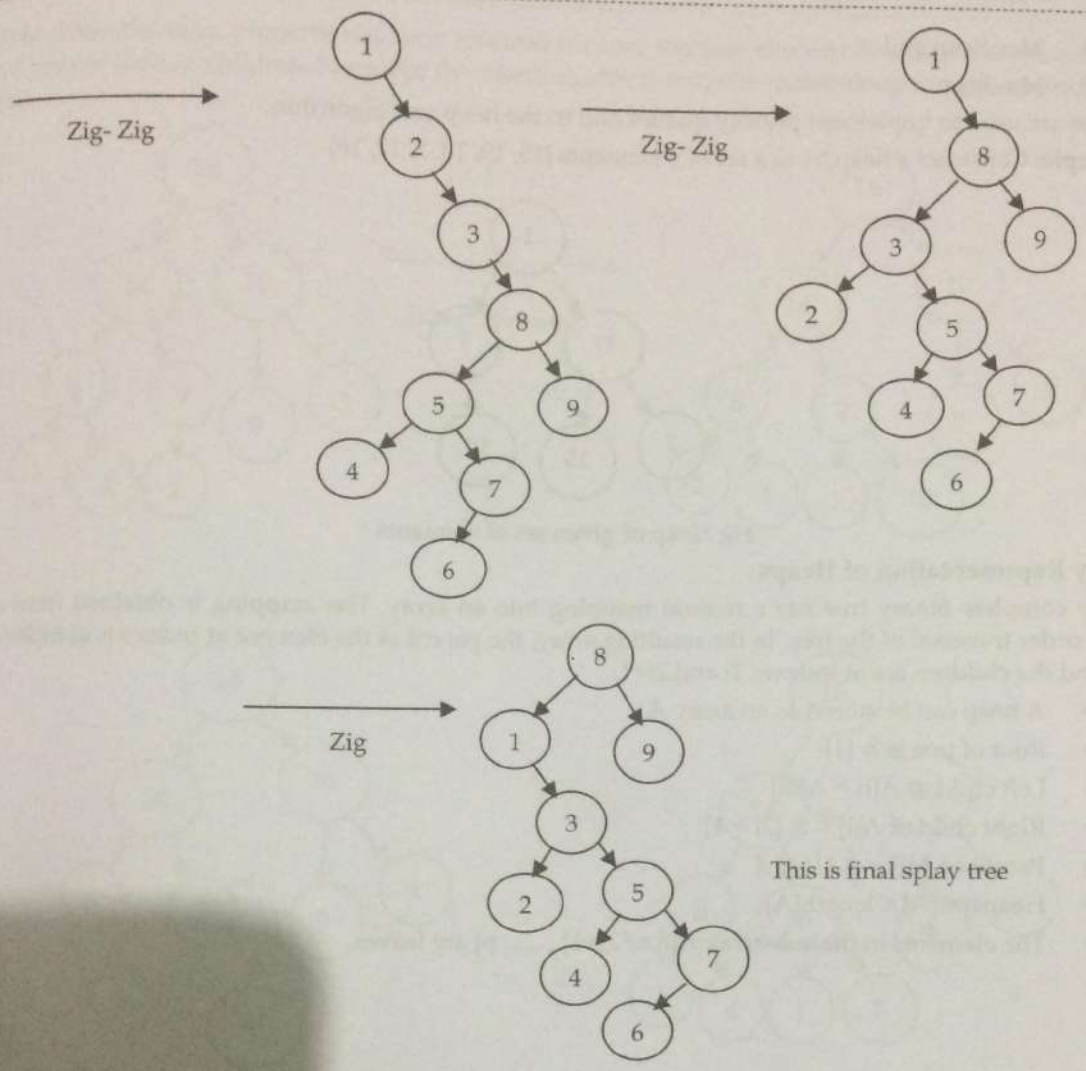
Example 1: Splay at node 8 of following tree



Solution:

Zig- Zag





Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps:

1. Check whether tree is Empty.
2. If tree is Empty then insert the **newNode** as Root node and exit from the operation.
3. If tree is not empty then insert the newNode as a leaf node using Binary Search tree insertion logic.
4. After insertion, **Splay** the **newNode**

Heaps

A heap is an almost complete binary tree whose elements have keys that satisfy the following heap property: the keys along any path from root to leaf are descending (i.e. non-increasing). Heaps could represent family descendant trees because the heap property means that every parent is older than its children.

In brief, a heap is an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value in parent node. This type of heap is called max heap. By default, the heap is max heap. There are two types of heap:

1. Max heap and
2. Min heap

Heaps are used to implement priority queues and to the heap sort algorithm.

Example: Construct a heap from a set of 6 elements {15, 19, 10, 7, 17, 16}

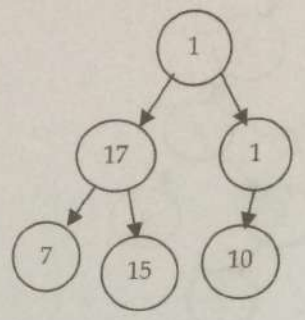
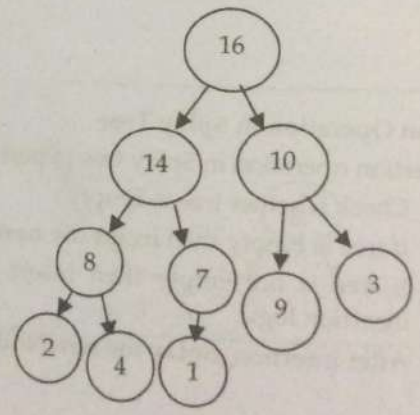


Fig Heap of given set of elements

Array Representation of Heaps

Every complete binary tree has a natural mapping into an array. The mapping is obtained from a level-order traversal of the tree. In the resulting array, the parent of the element at index *i* is at index *i* /2, and the children are at indexes 2*i* and 2*i*+1.

- A heap can be stored as an array *A*.
- Root of tree is A [1]
- Left child of A[i] = A[2i]
- Right child of A[i] = A [2i + 1]
- Parent of A[i] = A [$\lfloor i/2 \rfloor$]
- Heapsize[A] ≤ length[A]
- The elements in the sub-array A[($\lfloor n/2 \rfloor$+1) ..... n] are leaves



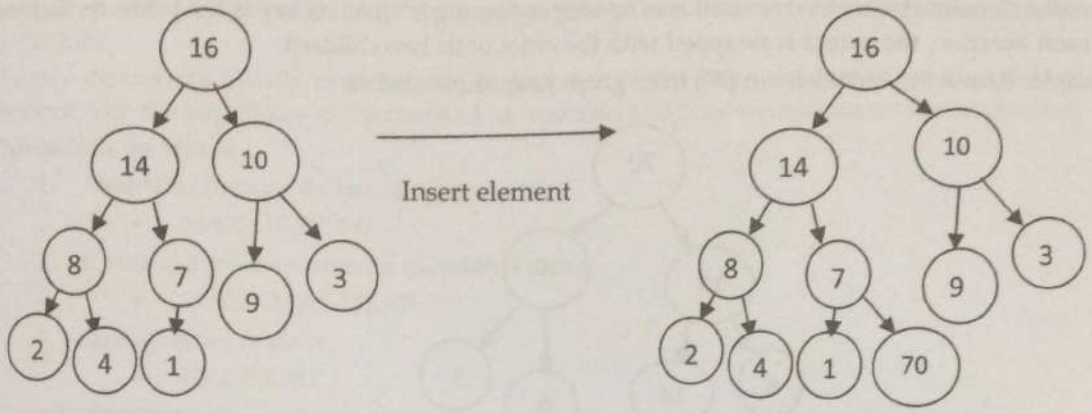
Inserting element to an existing heap

Elements are inserted into a heap next to its right-most leaf at the bottom level. Then the heap property is restored by percolating the new element up the tree until it is no longer "older" (i.e., its key is greater) than its parent. On each iteration, the child is swapped with its parent.

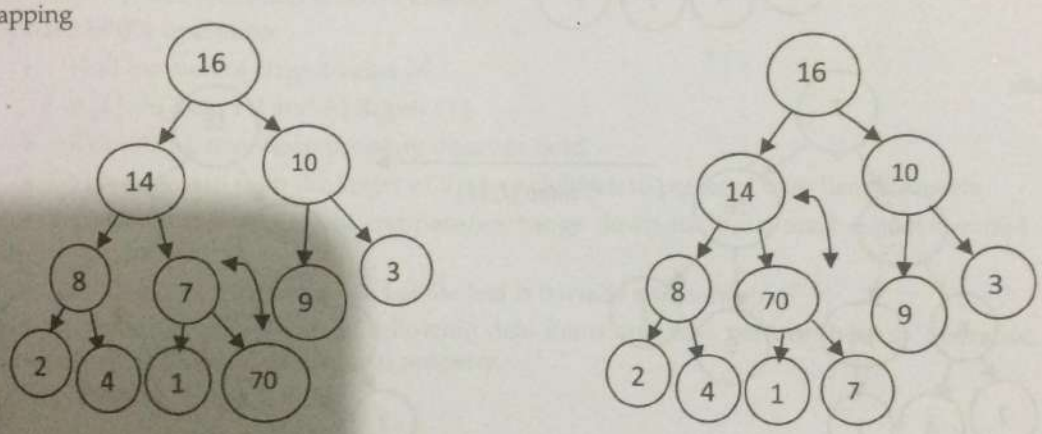
Example: Insert element 70 to given heap

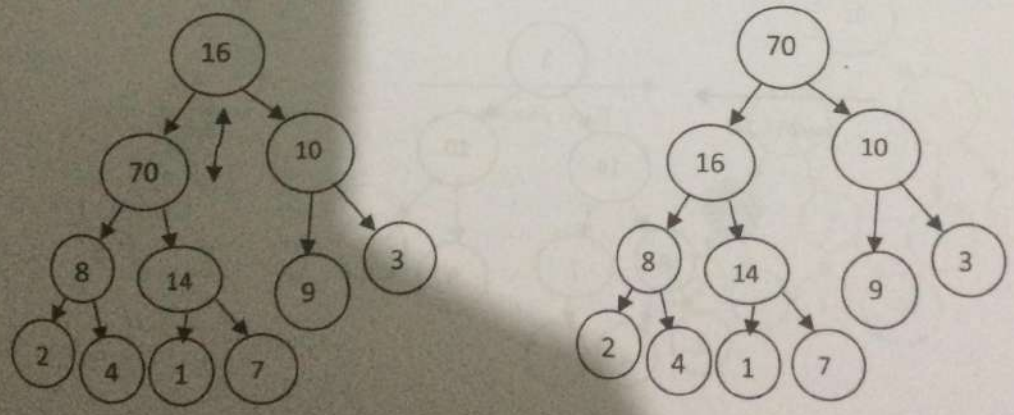
Figure below shows how the key 70would be inserted into the heap. The element 70 is added to the tree as a new last leaf. Then it is swapped with its parent element 7 because 70 > 7. Then it is swapped with its parent element 14because 70 > 14, again swap it with its parent element 16 because

70>16. Now the heap property has been restored because the new element 70 is less than its parent and greater than its children. Note that the insertion affects only the nodes along a single root-to-leaf path.



Swapping

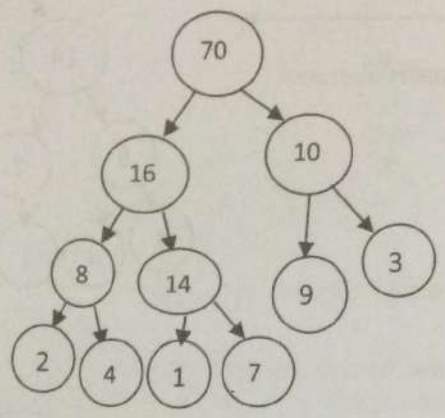




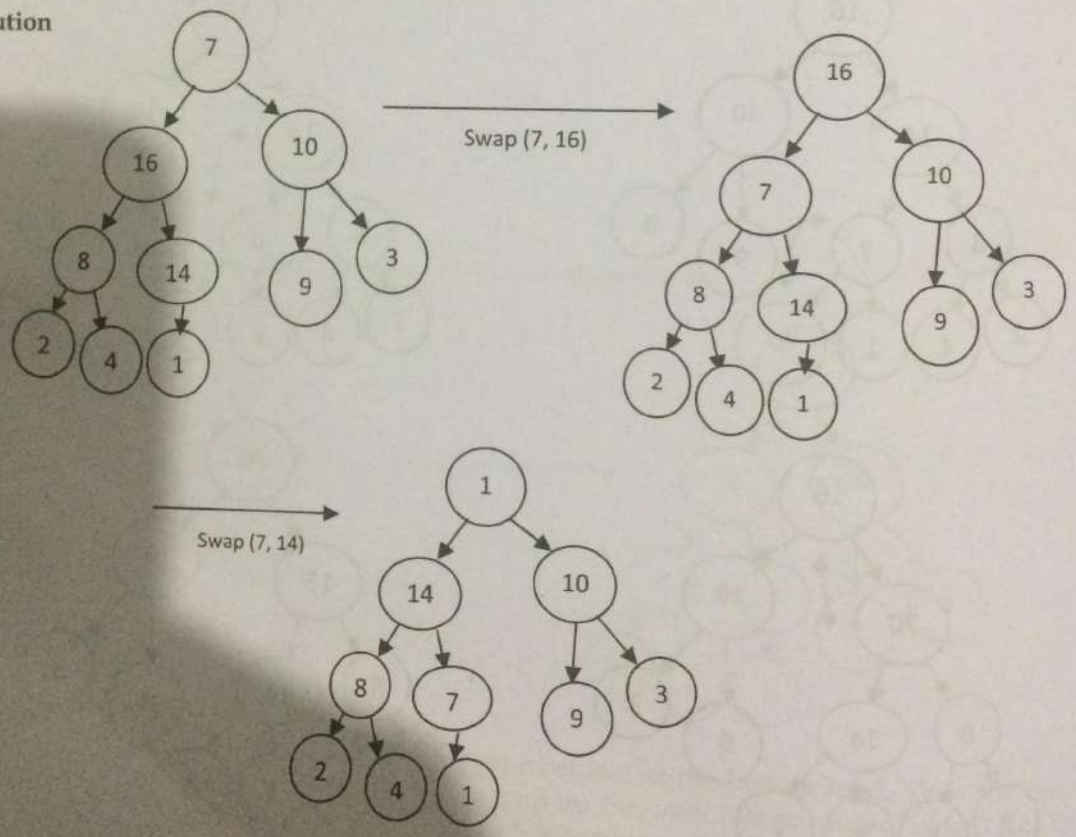
Deleting element from an existing heap

The heap removal algorithm always removes the root element from the tree. This is done by moving the last leaf element into the root element and then restoring the heap property by per collating the new root element down the tree until it is no longer "younger" (i.e., its key is less) than its children. On each iteration, the parent is swapped with the older of its two children.

Example: Removing root element (70) from given heap shown below



Solution



Heaps as Priority Queues

A **stack** is a LIFO container: The last one in comes out first. A **queue** is a "FIFO container: The first one in comes out first. A **priority queue** is a "BIFO container": The best one in comes out first. That

means that each element is assigned a priority number, and the element with the highest priority comes out first. Priority queues are widely used in computer systems. For example, if a printer is shared by several computers on a local area network, the print jobs that are queued to it would normally be held temporarily in a priority queue wherein smaller jobs are given higher priority over larger jobs.

Priority queues are usually implemented as heaps since the heap data structure always keeps the element with the largest key at the root and its insertion and removal operations are so efficient.

Operations on Heaps

1. Maintain/Restore the max-heap property
   - MAX-HEAPIFY
2. Create a max-heap from an unordered array
   - BUILD-MAX-HEAP
3. Sort an array in place
   - HEAPSORT

Heapify Property

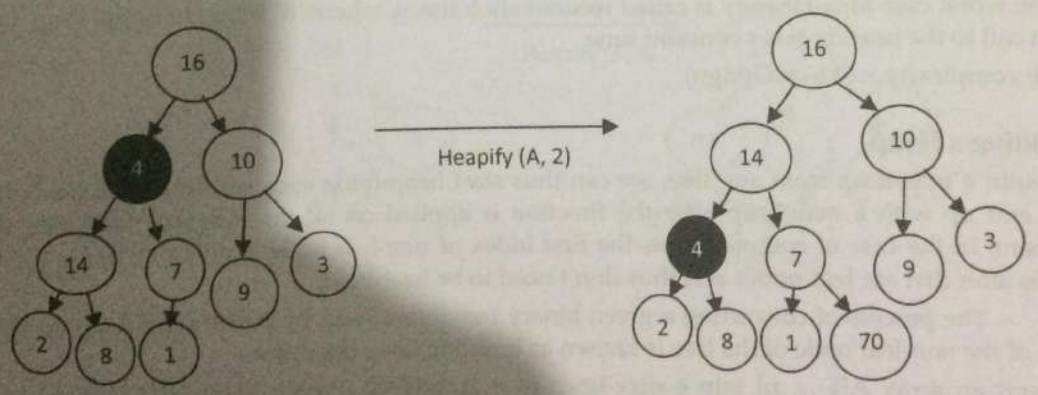
If any node violets the heap property then swap this node with its larger children to maintain the heap property, this operation is called heapify.

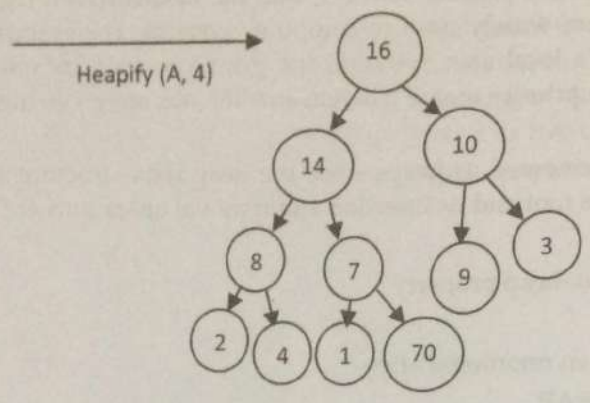
MAX-HEAPIFY operation

- Find location of largest value of:
  A[i], A[Left(i)] and A[Right(i)]
- If not A[i], max-heap property does not hold.
- Exchange A[i] with the larger of the two children to preserve max-heap property.
- Continue this process of compare/exchange down the heap until sub-tree rooted at *i* is a max-heap.
- At a leaf, the sub-tree rooted at the leaf is trivially a max-heap.

Example: construct binary tree of following data items and then perform heapify operation on the particular nodes that violets the heap property.

A[ ]={16, 4, 10, 14, 7, 9, 3, 2, 8, 1}





Algorithm

```
Max-Heapify(A, i, n)
{
        l = Left(i)
        r = Right(i)
        largest=i;
        if l ≤ n and A[l] > A[largest]
                largest = l
        if r ≤ n and A[r] > A[largest]
                largest = r
        if largest ≠ i
                exchange (A[i], A[largest])
        Max-Heapify(A, largest, n)
}
```

Analysis

In the worst case Max-Heapify is called recursively h times, where 'h' is height of the heap and since each call to the heapify takes constant time

Time complexity = O(h) = O(logn)

Building a Heap

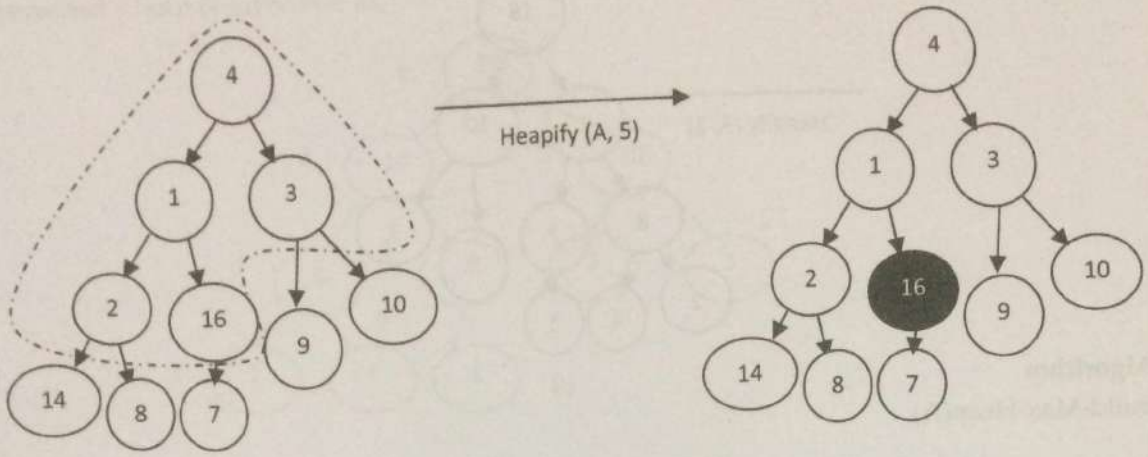
To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied on all the elements including the root element. In the case of complete tree, the first index of non-leaf node is given by n/2 - 1. All other nodes after that are leaf-nodes and thus don't need to be heapified.

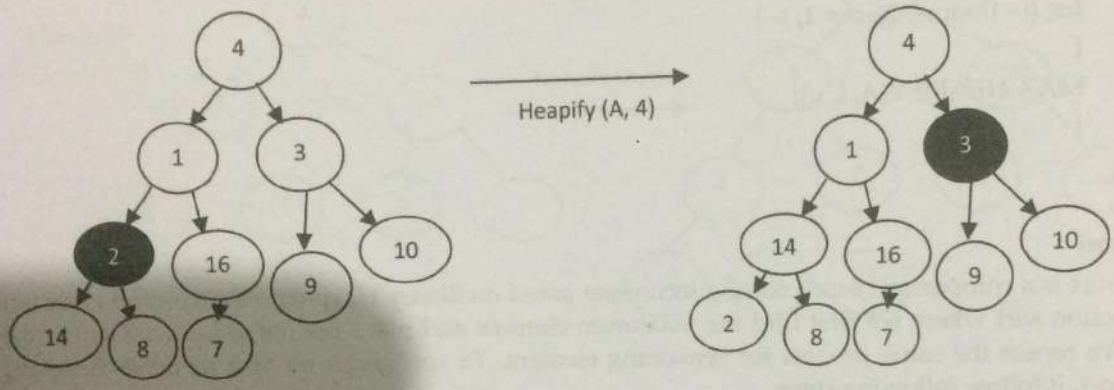
The process of converting a given binary tree into a heap by performing heap operation on each of the non-leaf node of the tree is known as building heap operation.

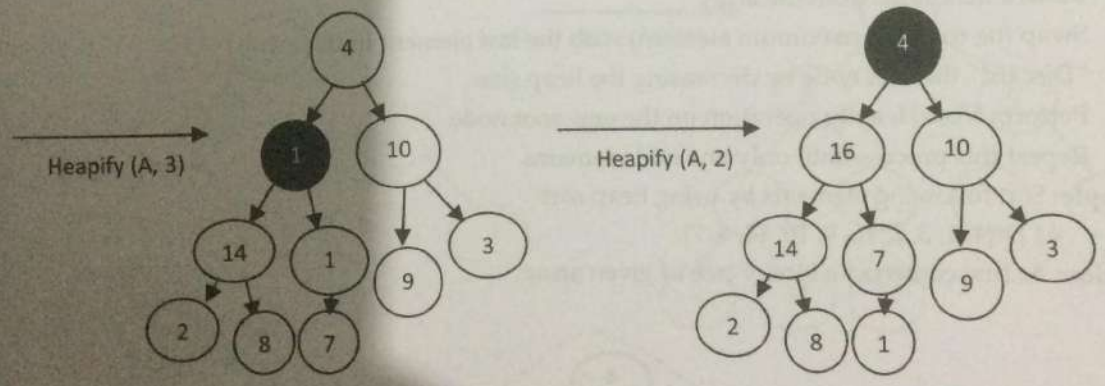
Convert an array A[1 ... n] into a max-heap (n = length[A]). Apply MAX-HEAPIFY on elements between 1 and floor function of (n/2).

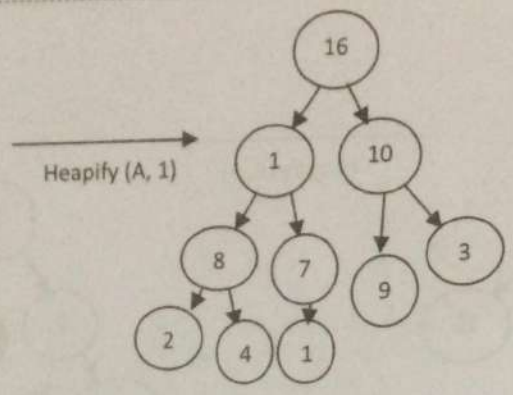
Example: At first Construct binary tree of following data items and then build heap by performing heapify operation on every non-leaf nodes.

A[ ]={4, 1, 3, 2, 16, 9, 10, 14, 8, 7}


Heapify (A, 5)


Heapify (A, 4)


Heapify (A, 3)
Heapify (A, 2)



Algorithm

```
Build-Max-Heap(A)
{
        n = length[A]
        for (i= floor(n/2); i>= 1; i--)
        {
        MAX-HEAPIFY(A, i, n);
        }
}
```

Heapsort

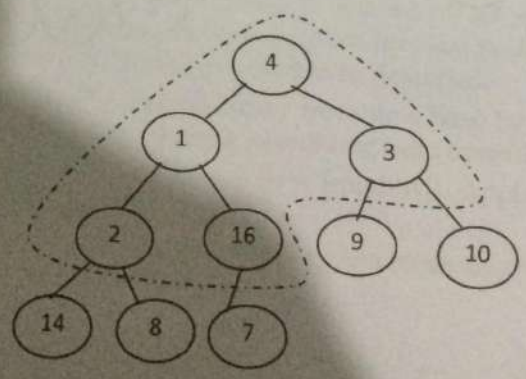
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element. To sort given set of n elements by using heap sort, it follow following steps

- Build a max-heap from the array
- Swap the root (the maximum element) with the last element in the array
- "Discard" this last node by decreasing the heap size
- Perform Max-Heapify operation on the new root node
- Repeat this process until only one node remains

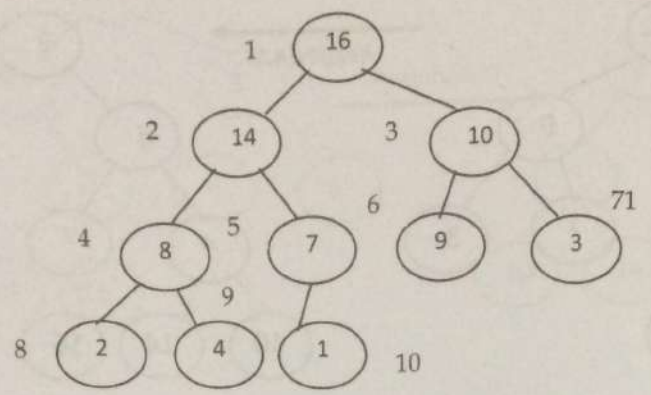
Example: Sort following elements by using heap sort

A[ ] ={4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

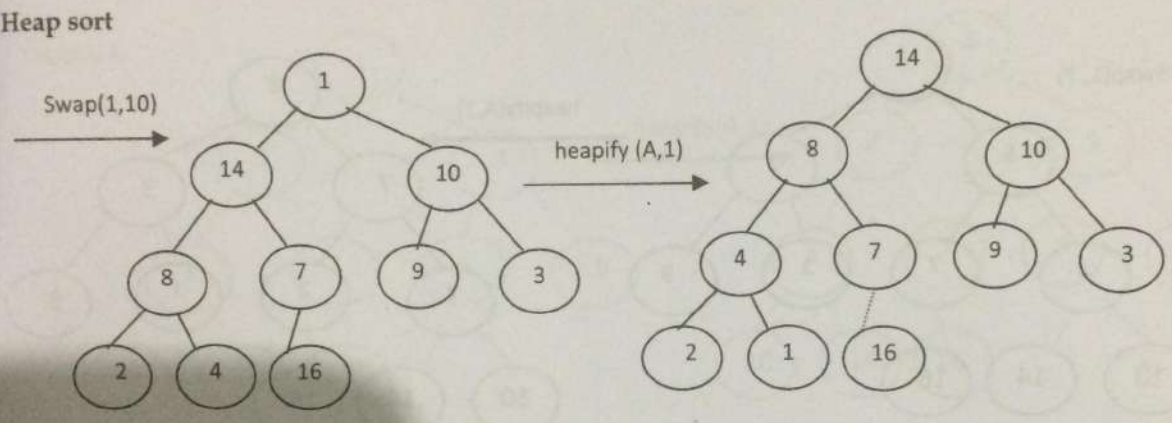
Solution: At first construct a binary tree of given array,

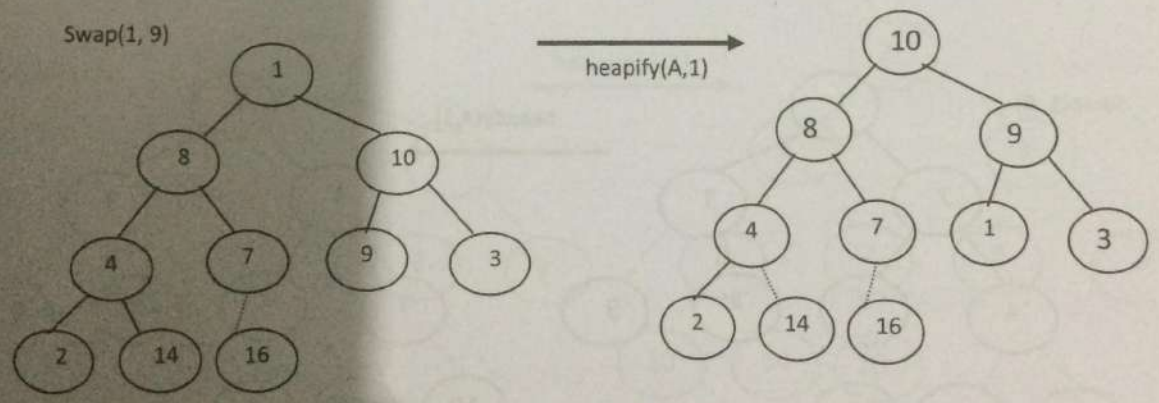


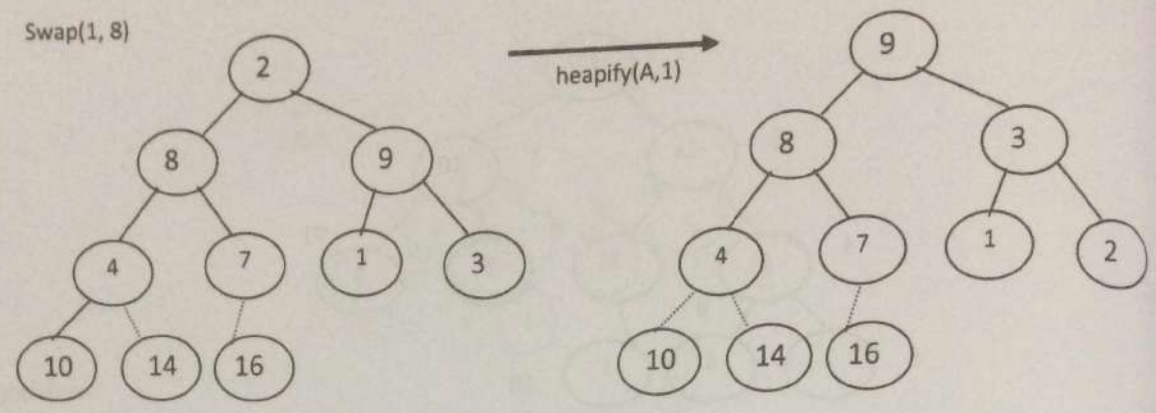
Now construct a heap of given tree as,

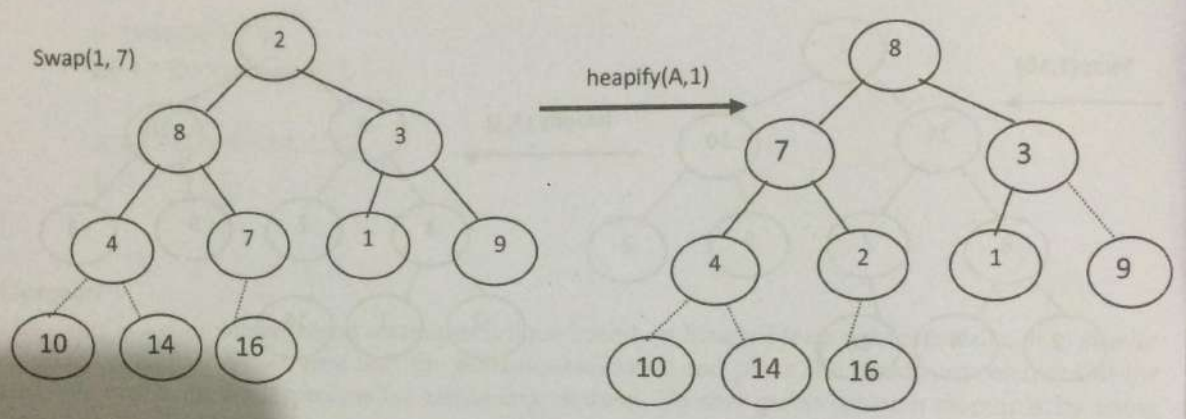


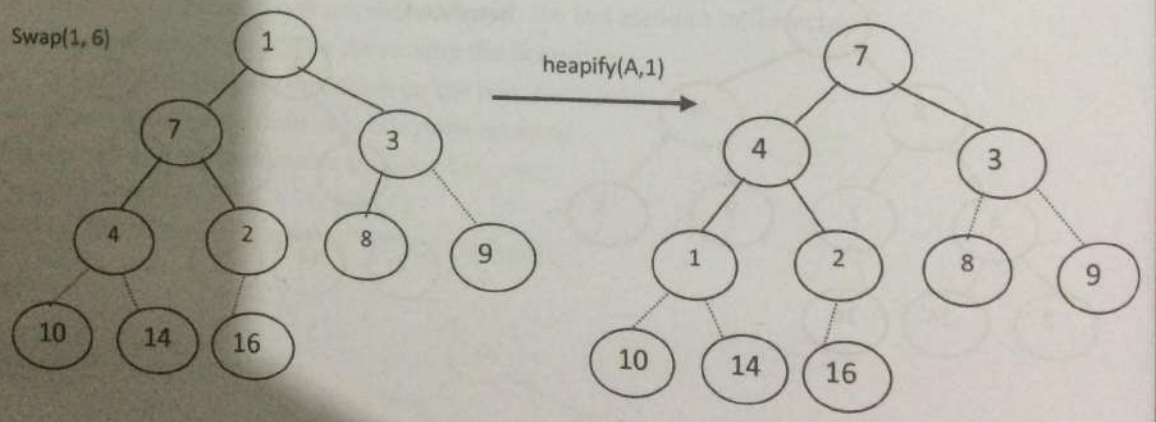
Heap sort



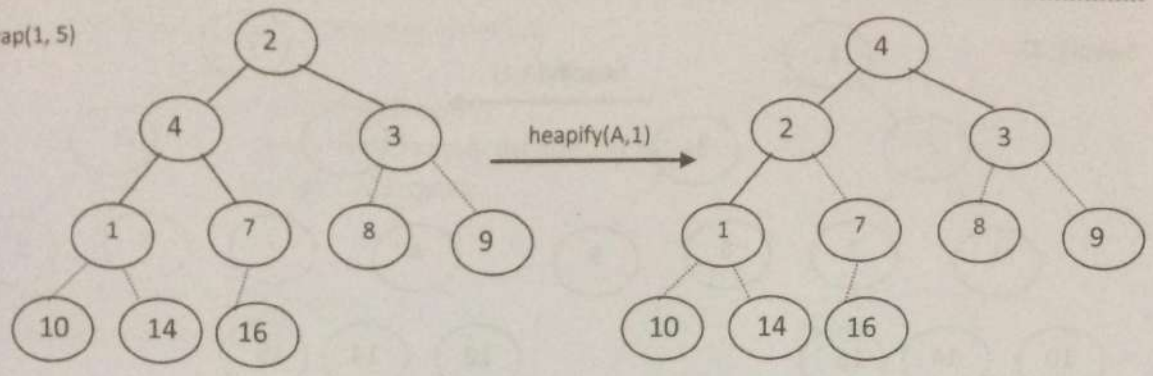



Swap(1, 8)
heapify(A,1)

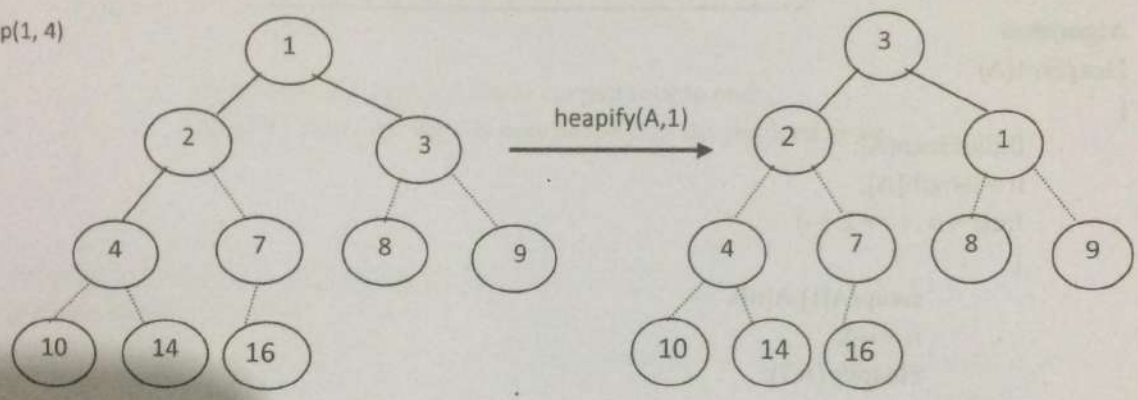

Swap(1, 7)
heapify(A,1)


Swap(1, 6)
heapify(A,1)

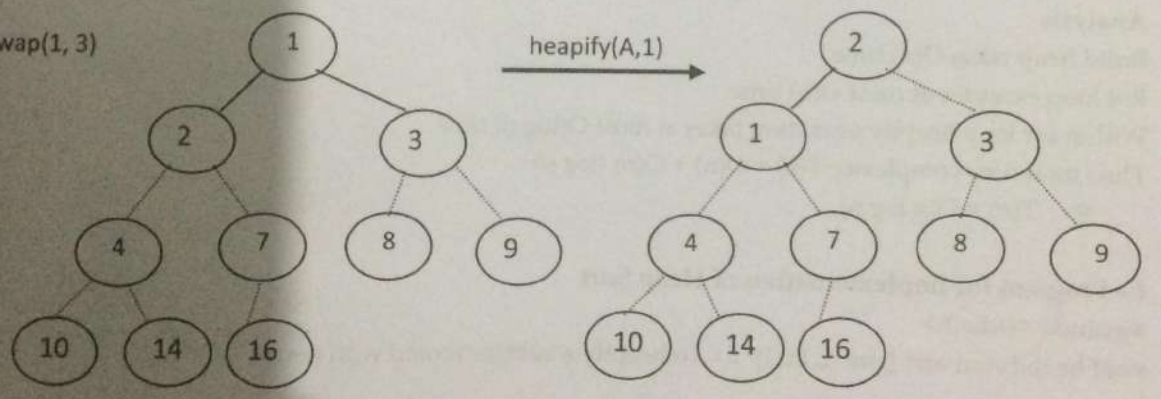
Swap(1, 5)

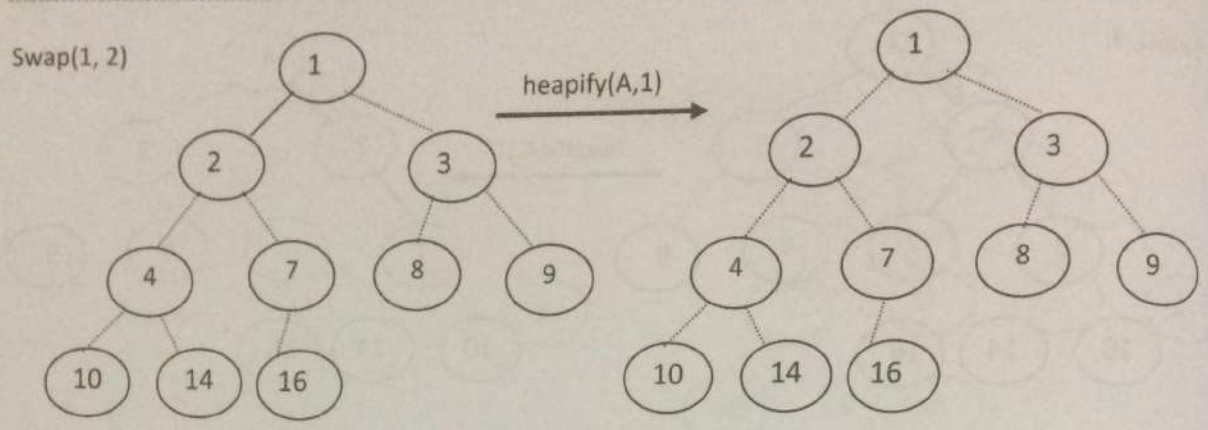


Swap(1, 4)



Swap(1, 3)





1	2	3	4	7	8	9	10	14	16

Algorithm

```
HeapSort(A)
{
        BuildHeap(A);
        n = length[A];
        for(i = n ; i >= 2; i--)
        {
                swap(A[1],A[n]);
                n = n-1;
                Heapify(A,1);
        }
}
```

Analysis

Build heap takes O(n) time

For loop executes at most O(n) time

Within for loop heapify operation takes at most O(log n) time

Thus total time complexity T(n) = O(n) + O(n) (log n)

⇨ T(n) = O(n log n)

C- Program for implementation of Heap Sort

```
#include <stdio.h>
void heapify(int arr[ ], int n, int i)  // To heapify a subtree rooted with node i
{
        int largest = i; // Initialize largest as root
        int l = 2*i + 1; // left = 2*i + 1
        int r = 2*i + 2; // right = 2*i + 2
        if (l < n && arr[l] > arr[largest])      // If left child is larger than root
                largest = l;
        if (r < n && arr[r] > arr[largest])      // If right child is larger than largest so far
                largest = r;
```

```
        if (largest != i)    // If largest is not root
        {
                swap(arr[i], arr[largest]);
                // Recursively heapify the affected sub-tree
                heapify(arr, n, largest);
        }
    }
// main function to do heap sort
void heapSort(int arr[ ], int n)
{
        for (int i = n / 2 - 1; i >= 0; i--)
                Heapify(arr, n, i);
        for (int i=n-1; i>=0; i--)
        {
                swap(arr[0], arr[i]);  // Move current root to end
                Heapify(arr, i, 0);  // call max heapify on the reduced heap
        }
}

/* A utility function to print array of size n */
void printArray(int arr[ ], int n)
{
        for (int i=0; i<n; ++i)
          cout << arr[i] << " ";
        cout << "\n";
}

int main()
{
        int arr[ ] = {12, 11, 13, 5, 6, 7};
        int n = 6;
        HeapSort(arr, n);
        Printf("Sorted array is:\n");
        printArray(arr, n);
}
```

Output:

Sorted array is:
5 6 7 11 12 13

Exercise

1. How many leaf nodes do the full binary tree of height $h = 3$ have?
2. How many internal nodes do the full binary tree of height $h = 3$ have?
3. How many leaf nodes do a full binary tree of height $h = 9$ have?
4. How many internal nodes do a full binary tree of height $h = 9$ have?
5. How many nodes do a full binary tree of height $h = 9$ have?
6. What is the range of possible heights of a binary tree with $n = 100$ nodes?
7. What are the two main applications of heaps? How it is differ from BST?
8. How efficient are insertions into and removals from a heap?
9. Why a priority queue is called a BIFO container?
10. What is splay tree? Describe its importance.
11. What do you mean by rotation operation in AVL tree?
12. What is the difference between a queue and a priority queue?
13. Why are heaps used to implement priority queues?
14. If the nodes of a binary tree are numbered according to their natural mapping, and the visit operation prints the node's number, which traversal algorithm will print the numbers in order?
15. Draw the expression tree for a*(b + c)*(d*e + f).
16. What are the bounds on the number n of nodes in a binary tree of height 4?
17. What are the bounds on the height h of a binary tree with 7 nodes?
18. What form does the highest binary tree have for a given number of nodes?
19. What form does the lowest binary tree (i.e., the least height) have for a given number of nodes?
20. The order of visitation in the binary tree traversal:
    a. Level order: A, B, C, D, E, F, G, H, I, J, K
    b. Preorder: A, B, D, E, H, I, C, F, J, G, K
    c. In order: D, B, H, E, I, A, F, J, C, G, K
    d. Post order: D, H, I, E, B, J, F, K, G, C, A

•••

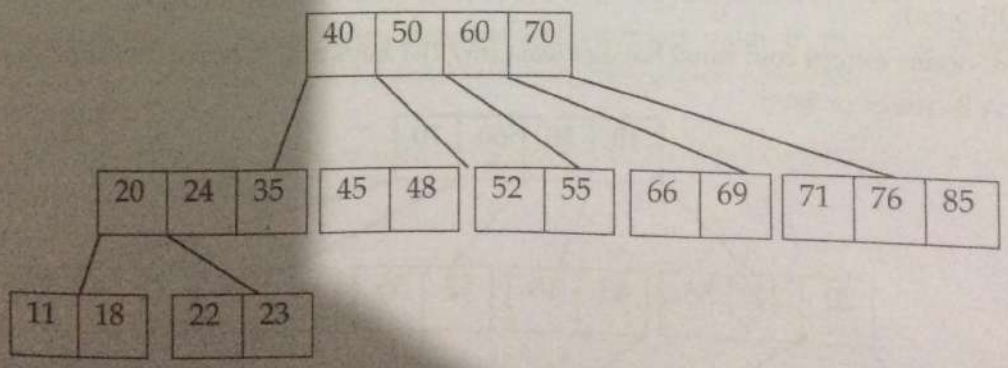
Chapter

8

Multi-way Trees

A **multiway tree** is a tree that can have more than two children. A **multiway tree of order m** or an **m-way tree** is one in which a tree can have m children. As with the other trees that have been studied, the nodes in an m-way tree will be made up of key fields, in this case m-1 key fields and pointers to children. If $k \leq m$ is the number of children, then the node contains exactly $k - 1$ keys, which partition all the keys into k subsets consisting of all the keys less than the first key in the node, all the keys between a pair of keys in the node, and all keys greater than the largest key in the node.

Example: multi way tree of order 5



To make the processing of m-way trees easier some type of order will be imposed on the keys within each node, resulting in a **multi way search tree of order m** (or an **m-way search tree**). By definition an m-way search tree is a m-way tree in which:

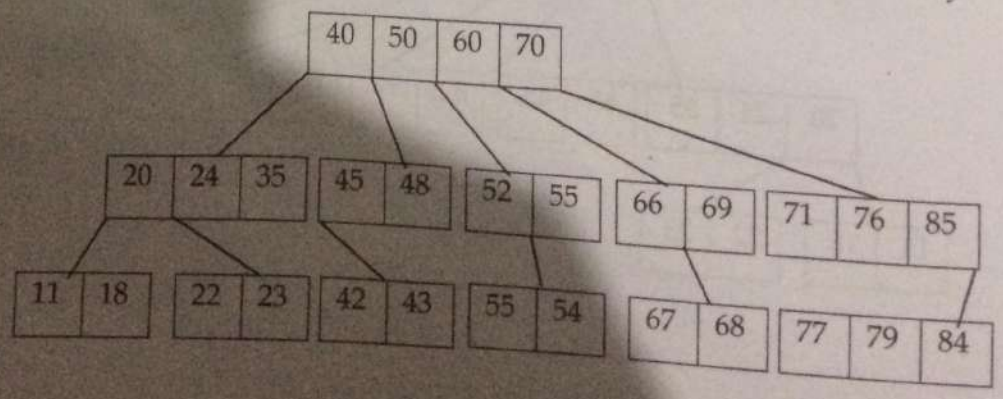
- Each node has m children and m-1 key fields
- The keys in each node are in ascending order.
- The keys in the first i children are smaller than the ith key
- The keys in the last m-i children are larger than the ith key

B-Trees

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees like AVL and Red-Black Trees, it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations such as search, insert, delete, find max, find min etc. require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree etc. A **B-tree of order m** is a multiway search tree with the following properties:

- All leaves are at same level.
- A B-Tree is defined by the term minimum degree't'. The value of t depends upon disk block size.
- Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
- All the key values within a node must be in Ascending Order.
- All nodes including root may contain at most (2t – 1) keys.
- If the root node is a non leaf node, then it must have at least 2 children
- Number of children of a node is equal to the number of keys in it plus 1.
- All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Log n).
- All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys

Example: A B-tree of order 5



Node creation in B-tree

A node of a B-tree is usually implemented as a structure containing an array of m – 1 cell for keys, an m-cell array of references (links) to other nodes, and possibly other information facilitating tree maintenance, such as the number of keys in a node and a leaf/nonleaf flag.

```
#define MAX 4
#define MIN 2
struct BtreeNode
{
    int val[MAX + 1];
    int count;
    struct BtreeNode *link[MAX + 1];
};
struct BtreeNode *root;
```

Operations on a B-Tree

The following operations are performed on a B-Tree:

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in B-Tree

In a B-Tree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows:

1. Start
2. Read the search element from the user
3. Compare, the search element with first key value of root node in the tree.
4. If both are matching, then display "Given node found!!!" and terminate the function
5. If both are not matching, then check whether search element is smaller or larger than that key value.
6. If search element is smaller, then continue the search process in left subtree.
7. If search element is larger, then compare with next key value in the same node and repeat step 3, 4, 5 and 6 until we found exact match or comparison completed with last key value in a leaf node.
8. If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.
9. Stop

Searching a B-tree Algorithm

An algorithm for finding a key in B-tree is simple. Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node. Follow the appropriate pointer to a child node. Examine the key fields in the child node and continue to follow

the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value. An algorithm for finding a key in a B-tree is simple, and is coded as follows:

```
public BTreeNode BTreeSearch(int key)
{
        return BTreeSearch(key,root);
}
protected BTreeNode BTreeSearch(int key, BTreeNode node)
{
        if (node != null)
        {
                int i = 1;
                for ( ; i <= node.keyTally && node.keys[i-1] < key; i++);
                        if (i > node.keyTally || node.keys[i-1] > key)
                                return BTreeSearch(key,node.references[i-1]);
                        else
                                return node;
        }
        else
                return null;
}
```

Insertion Operation in B-Tree

In a B-Tree, the new element must be added only at leaf node. That means, always the new key value is attached to leaf node only. The insertion operation is performed as follows:

1. Start
2. Check whether tree is Empty or not.
3. If tree is Empty, then create a new node with new key value and insert into the tree as a root node.
4. If tree is Not Empty, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
5. If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
6. If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.
7. If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.
8. Stop

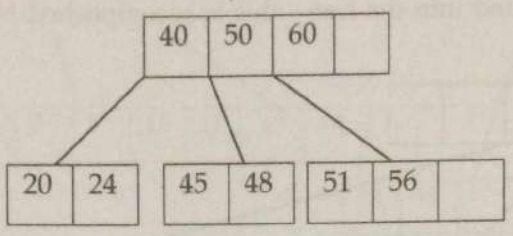
Insertion into a B-tree algorithm

The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-tree are not allowed to grow at their leaves; instead they are forced to grow at the root. When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:

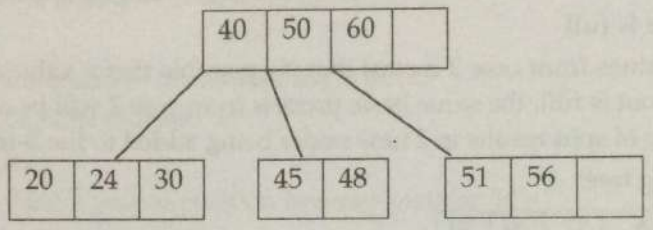
1. A key is placed into a leaf that still has room.
2. The leaf in which a key is to be placed is full.
3. The root of the B-tree is full.

Case 1: A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.



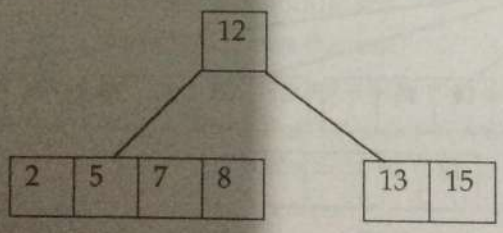
Inserting the number 30 results in:



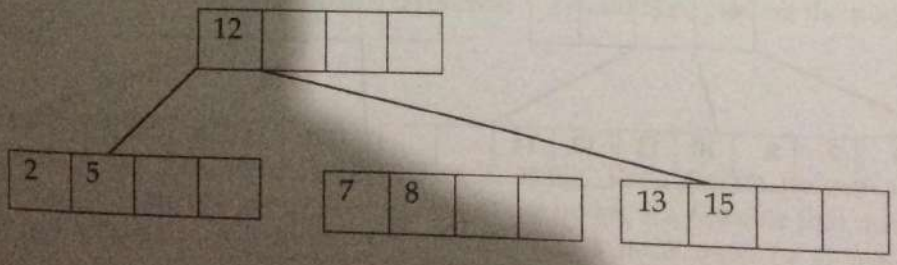
Case 2: The leaf in which a key is to be placed is full

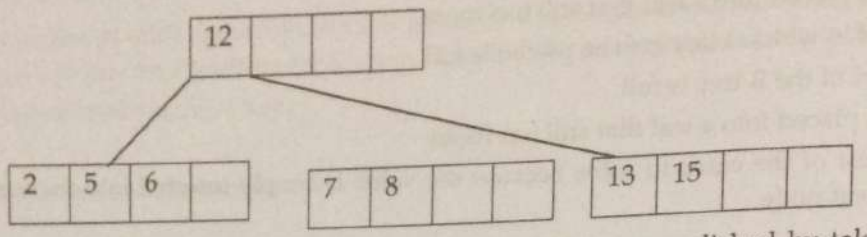
In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree. The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process is continues up the tree until all of the values have "found" a location.

Insert 6 into the following B-tree:

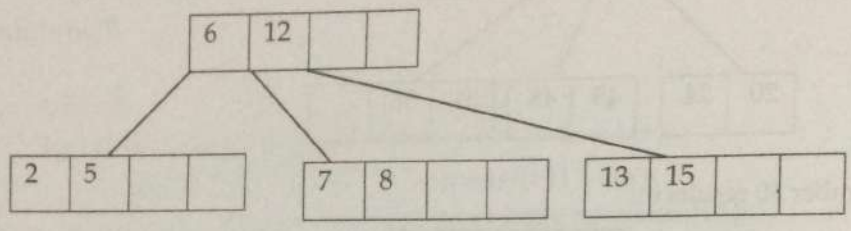


Results in a split of the first leaf node:





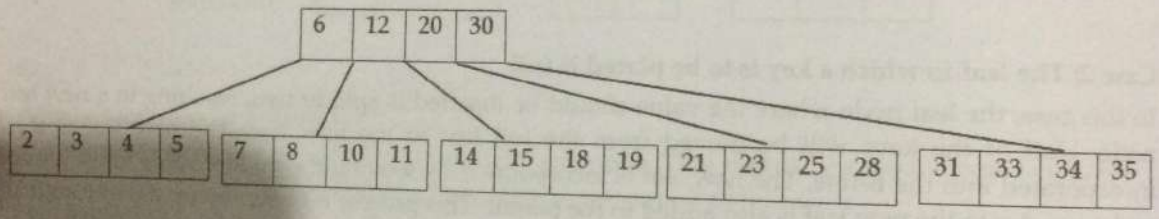
The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:

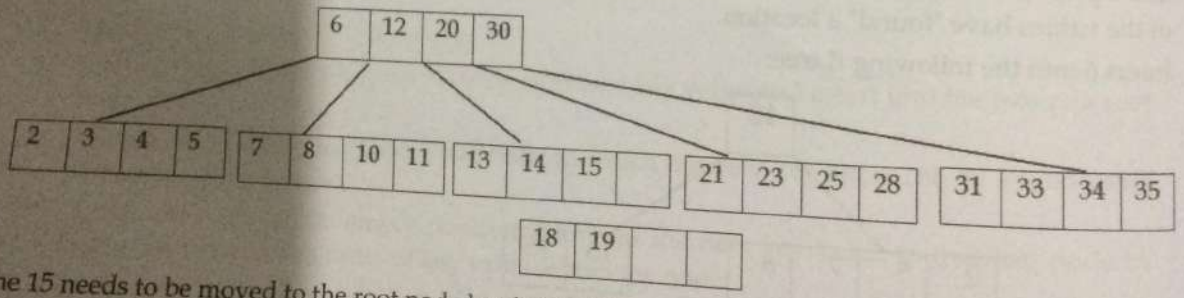


Case 3: The root of the B-tree is full

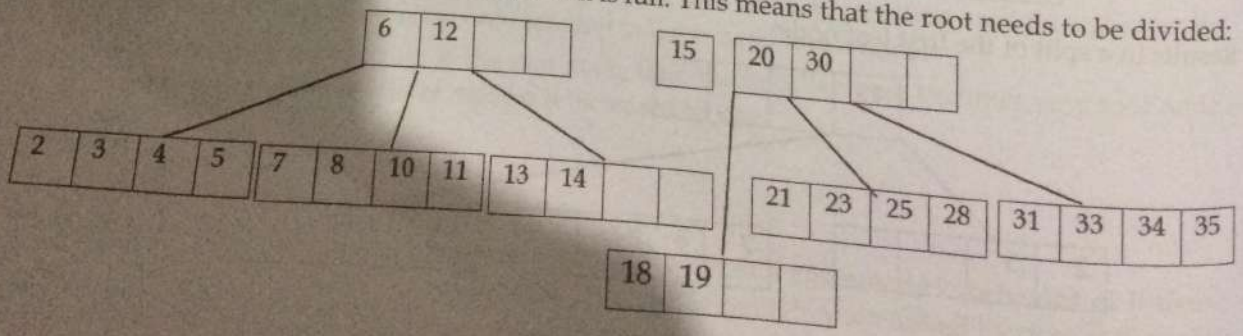
The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the B-tree.

Inserting 13 into the following tree:

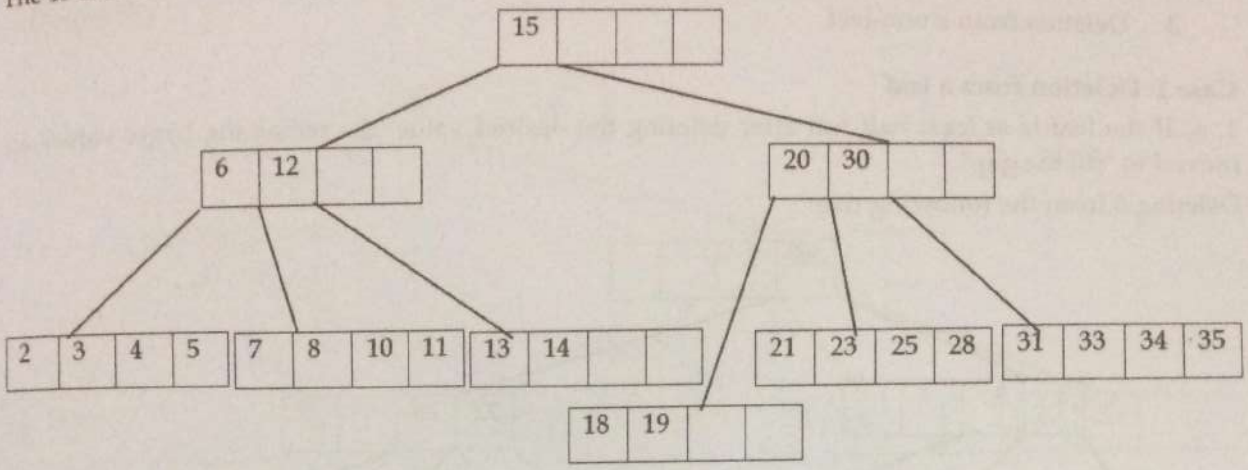




The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:



An algorithm for inserting keys in B-trees

1. Start
2. Read key element to be inserted say it be 'kl'
3. Find a leaf node to insert 'kl';
4. While (true)
   a. Find a proper position in array keys for 'kl';
   b. if node is not full
      i. Insert 'kl' and increment 'kl'
      ii. return;
   c. else
      i. Split node into node1 and node2; /*node1 = node, node2 is new; distribute keys and references evenly between node1 and node2 and initialize properly their key */
      ii. Set, kl = middle key
      iii. if node was the root
         - Create a new root as parent of node1 and node2;
         - Put 'kl' and references to node1 and node2 in the root, and set its key to 1;
         - return;
      iv. else
         - node = its parent; // and now process the node's parent;
5. Stop

Deleting from a B-tree

As usual, this is the hardest of the processes to apply. The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.

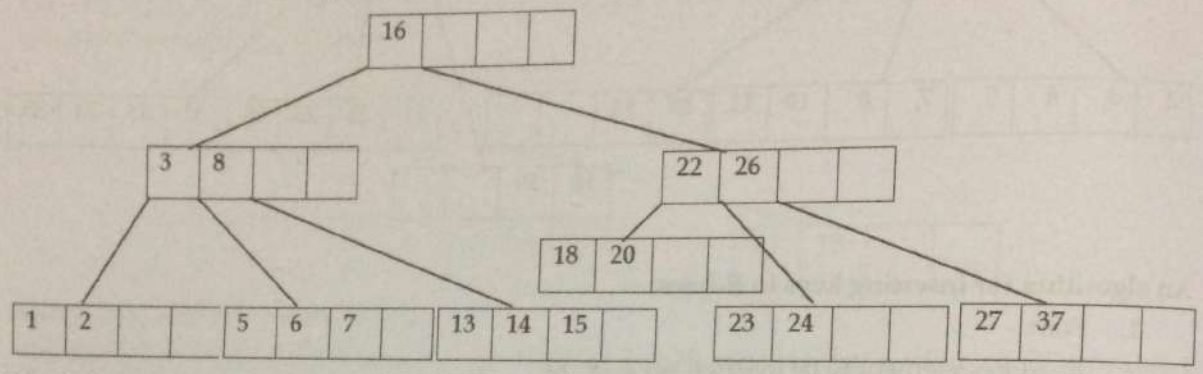
There are two main cases to be considered:

1. Deletion from a leaf
2. Deletion from a non-leaf

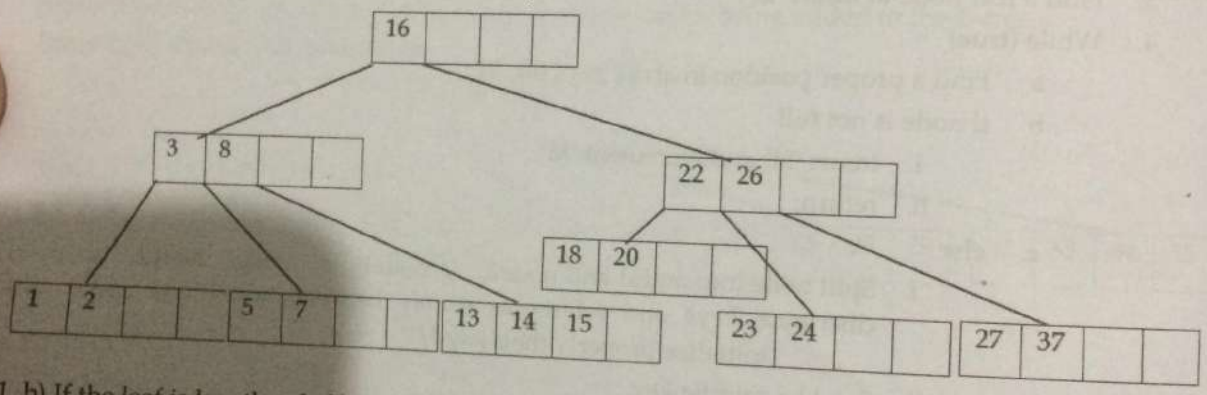
Case 1: Deletion from a leaf

1. a) If the leaf is at least half full after deleting the desired value, the remaining larger values are moved to "fill the gap".

Deleting 6 from the following tree:

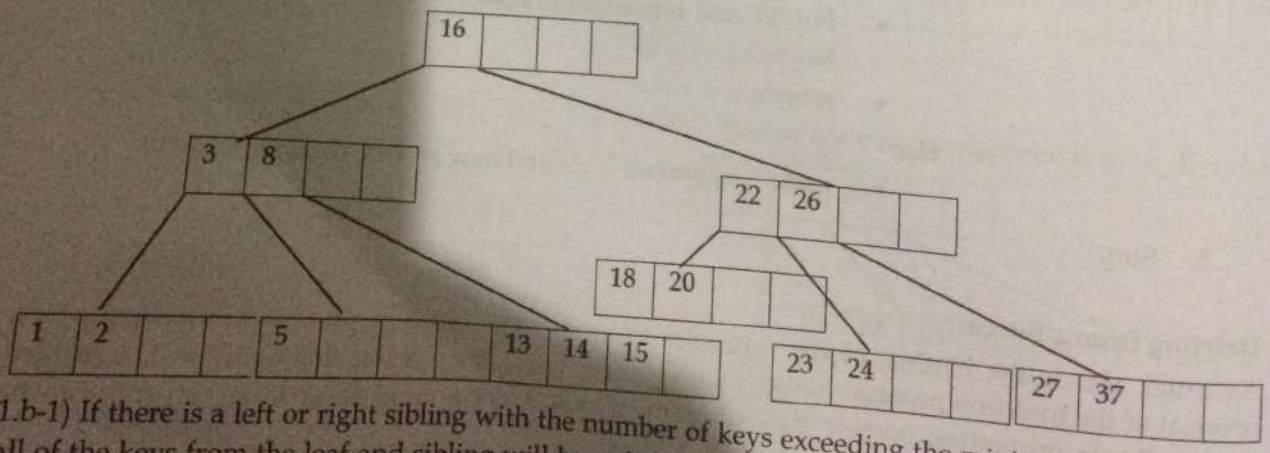


Results in:



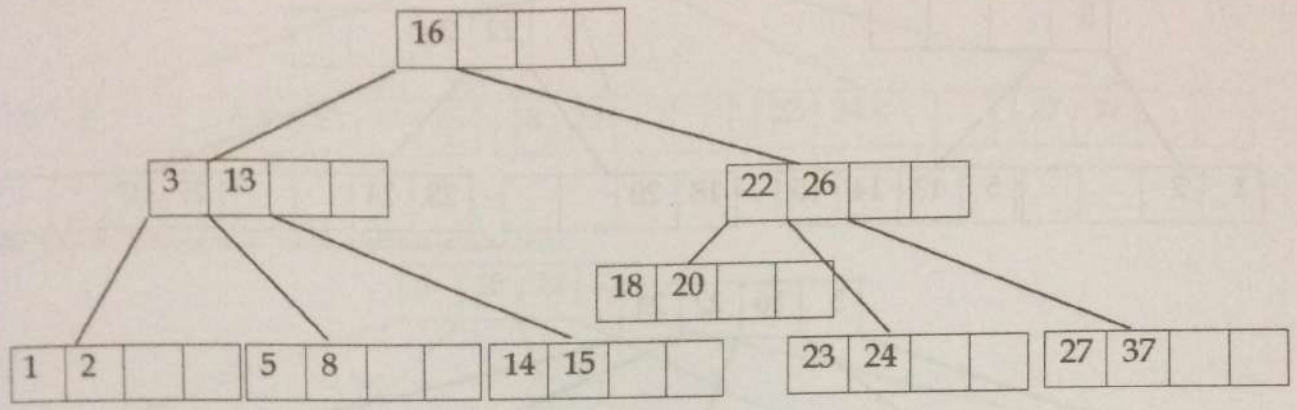
1. b) If the leaf is less than half full after deleting the desired value (known as underflow), two things could happen:

Deleting 7 from the tree above results in:

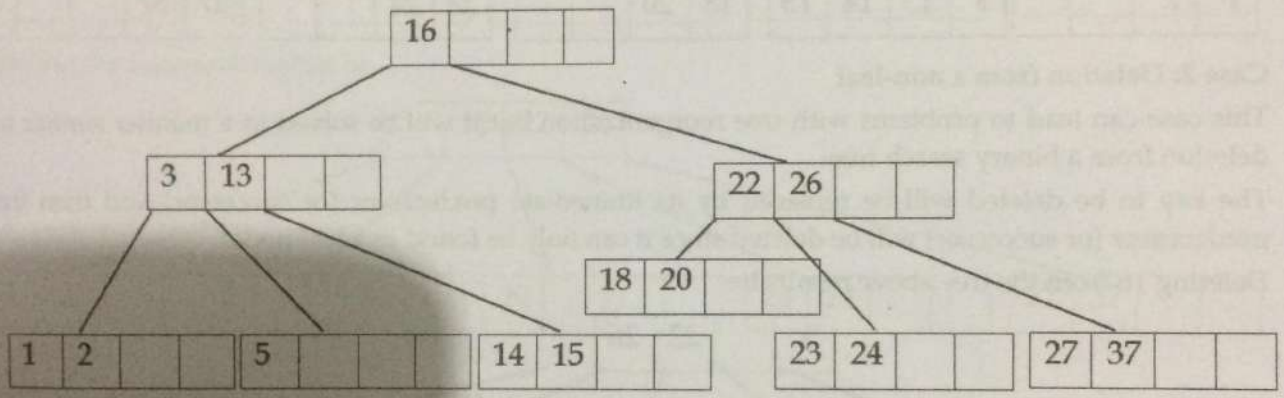


1.b-1) If there is a left or right sibling with the number of keys exceeding the minimum requirement, all of the keys from the leaf and sibling will be redistributed between them by moving the separator

key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.

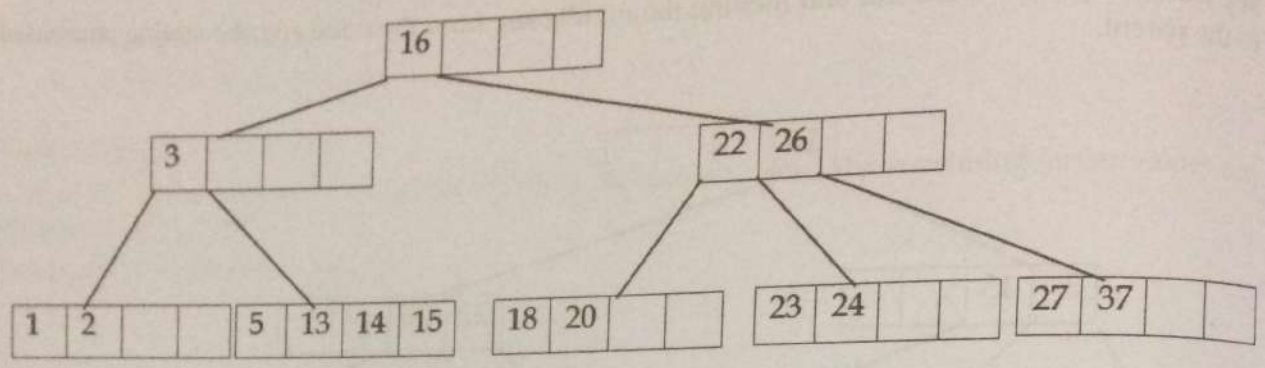


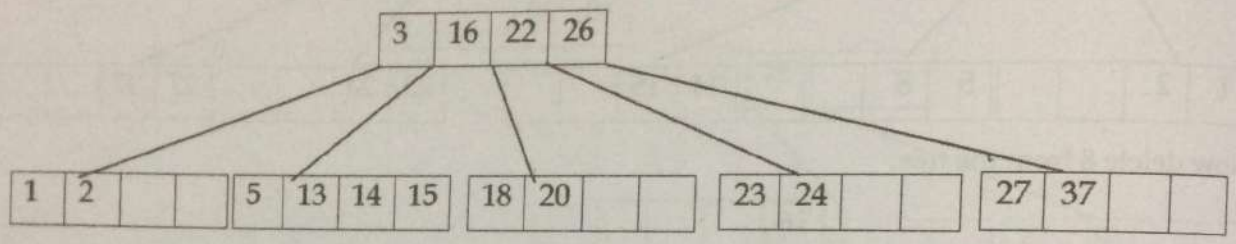
Now delete 8 from the tree:



1. b-2) If the number of keys in the sibling does not exceed the minimum requirement, then the leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf. The sibling node is discarded and the keys in the parent are moved to "fill the gap". It's possible that this will cause the parent to underflow. If that is the case, treat the parent as a leaf and continue repeating step (1. b-2) until the minimum requirement is met or the root of the tree is reached.

Special Case for 1.b-2: When merging nodes, if the parent is the root with only one key, the keys from the node, the sibling, and the only key of the root are placed into a node and this will become the new root for the B-tree. Both the sibling and the old root will be discarded.



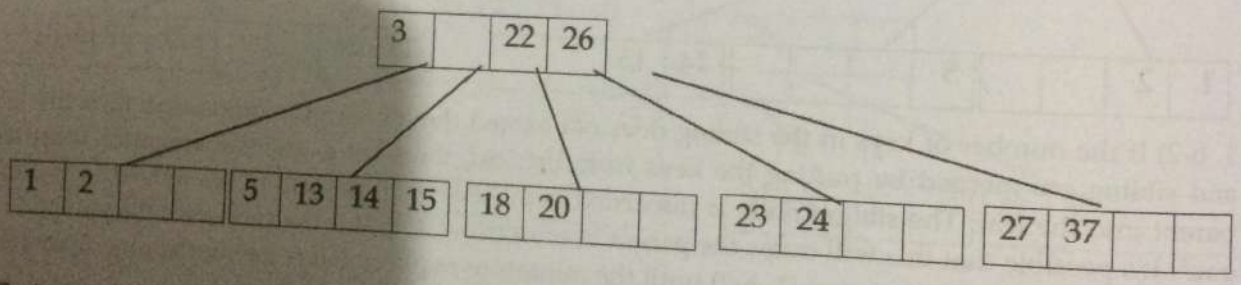


Case 2: Deletion from a non-leaf

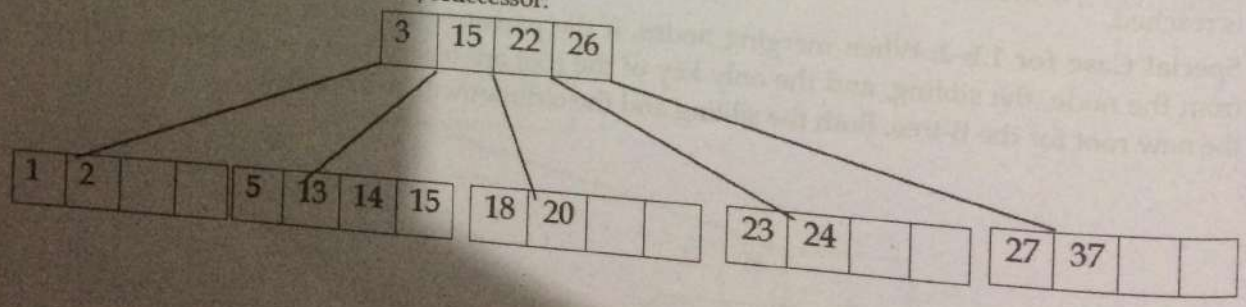
This case can lead to problems with tree reorganization but it will be solved in a manner similar to deletion from a binary search tree.

The key to be deleted will be replaced by its immediate predecessor (or successor) and then the predecessor (or successor) will be deleted since it can only be found in a leaf node.

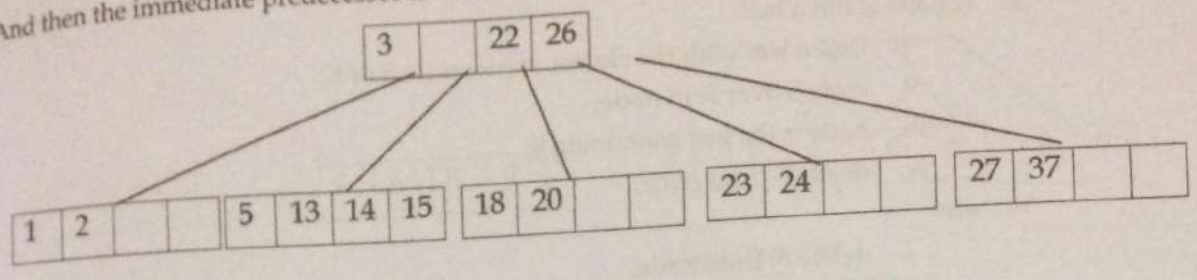
Deleting 16 from the tree above results in:



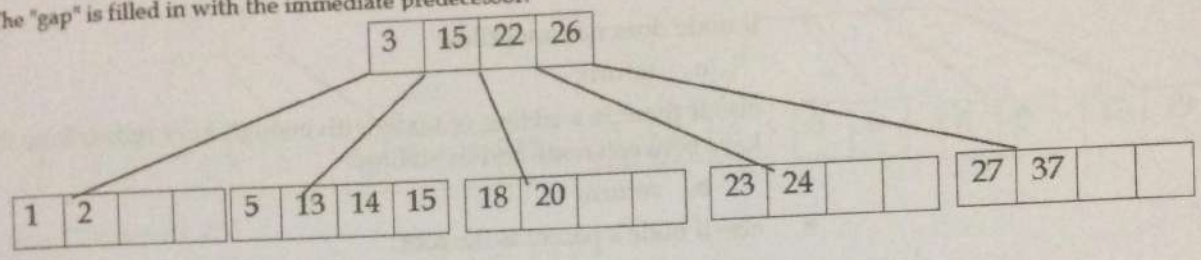
The "gap" is filled in with the immediate predecessor:



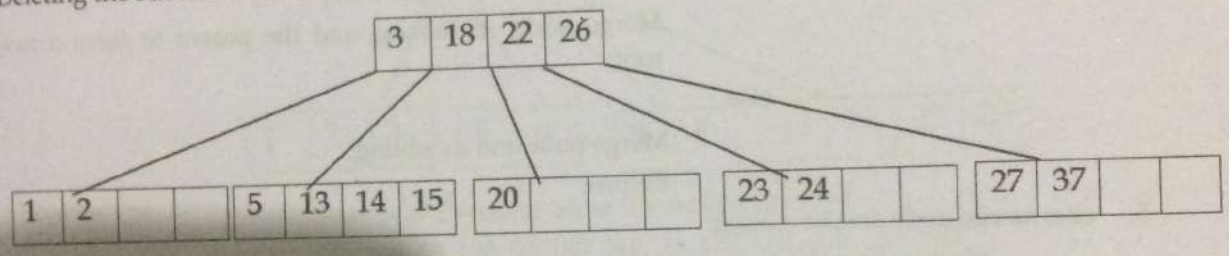
And then the immediate predecessor is deleted:



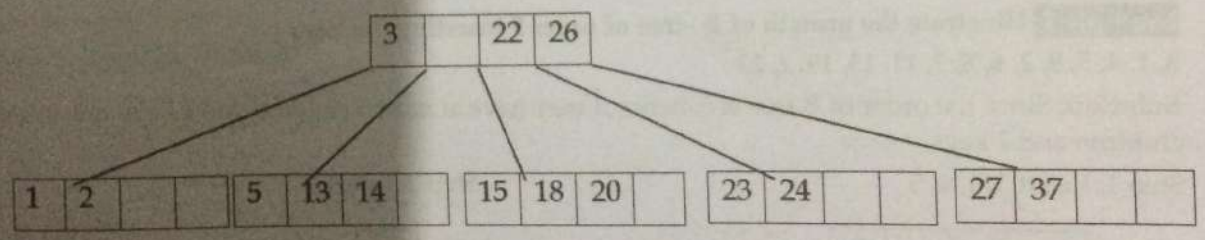
The "gap" is filled in with the immediate predecessor:



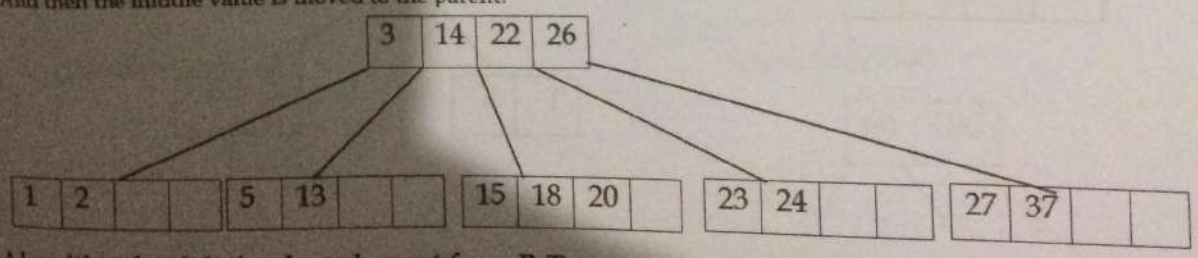
Deleting the successor results in:



The vales in the left sibling are combined with the separator key (18) and the remaining values. They are divided between the 2 nodes:



And then the middle value is moved to the parent:



Algorithm for deleting key element from B-Tree

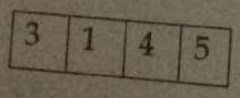
1. Start
2. Let K be the key element to be deleted
3. node = BTreeSearch(K, root);

4. if (node != null)
   a. if node is not a leaf
      i. find a leaf with the closest predecessor S of K;
      ii. copy S over K in node;
      iii. node = the leaf containing S;
      iv. delete S from node;
   b. else
      i. delete K from node;
      ii. while (true)
         - if node does not underflow
            a. return;
         - else if there is a sibling of node with enough keys redistribute the keys between node and its sibling;
            b. return;
         - else if node's parent is the root
            c. if the parent has only one key
               Merge node, its sibling, and the parent to form a new root;
         - else
            Merge node and its sibling;
            Return;
5. else merge node and its sibling;
   Node = its parent;
6. Stop

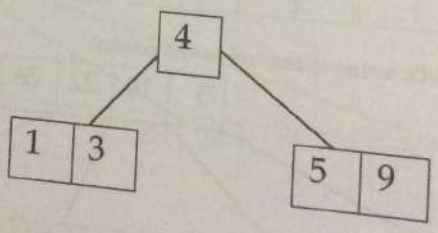
Example 1: Illustrate the growth of B –tree of order 5 inserting the keys 3, 1, 4, 5, 9, 2, 6, 8, 7, 11, 13, 19, 2, 23

Solution: Since the order of B-tree is 5, hence it may have at most 5 children and 4 keys and at least 3 children and 2 keys.

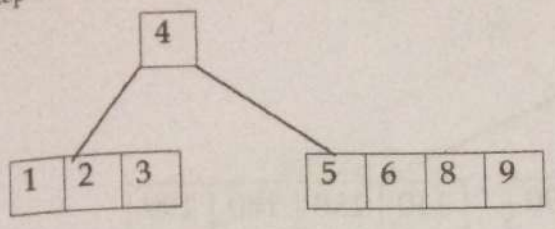
Step 1: Insert 3, 1, 4, 5



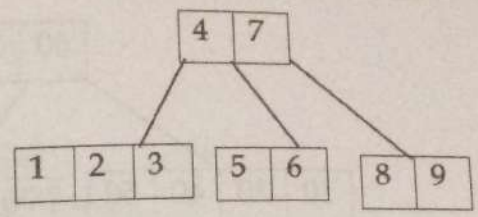
Step 2: Insert 9



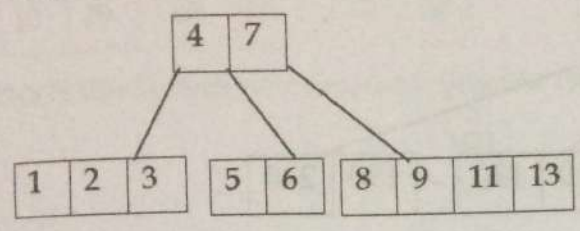
Step 3: Insert 2, 6, 8



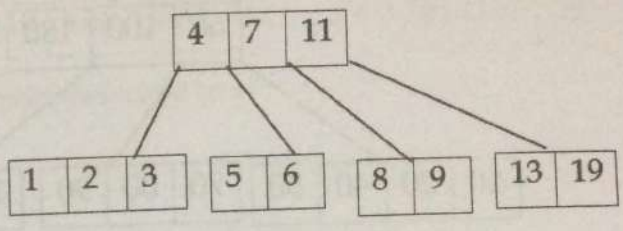
Step 4: Insert 7



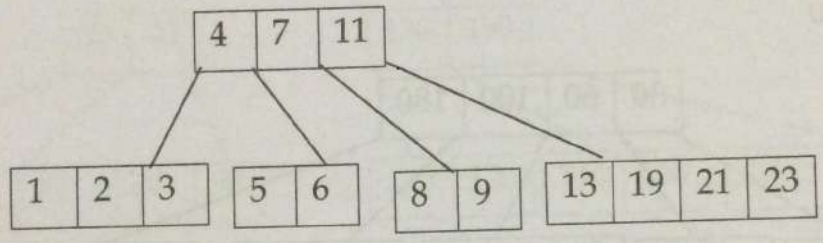
Step 5: Insert 11, 13



Step 6: Insert 19



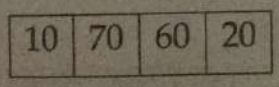
Step 7: Insert 21, 23



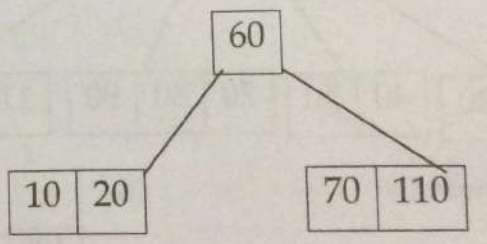
Example 2: Illustrate the growth of B –tree of order 5 inserting the keys
10, 70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90, 180, 240, 30, 120, 140, 200, 210, 160
And delete 80, 180, 160 and 40.

Solution: Since the order of B-tree is 5, hence it may have at most 5 children and 4 keys and at least 3 children and 2 keys.

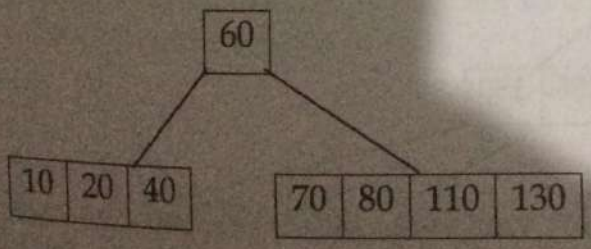
Step 1: Insert 10, 70, 60, 20



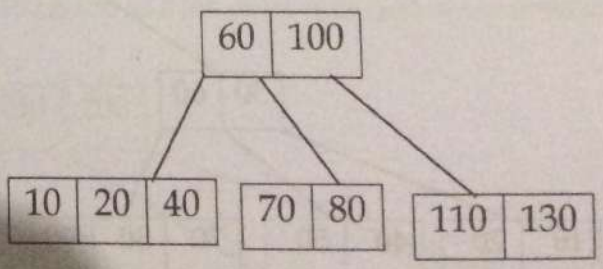
Step 2: Insert 110



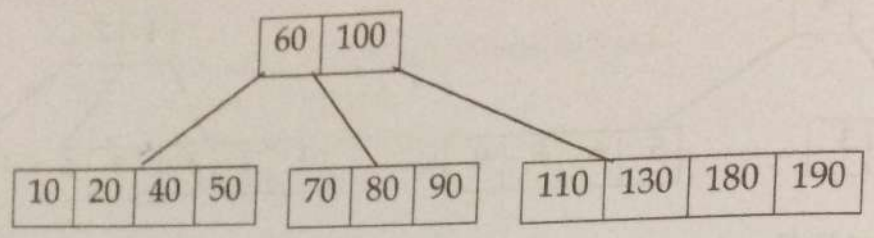
Step 3: Insert 40, 80, 130



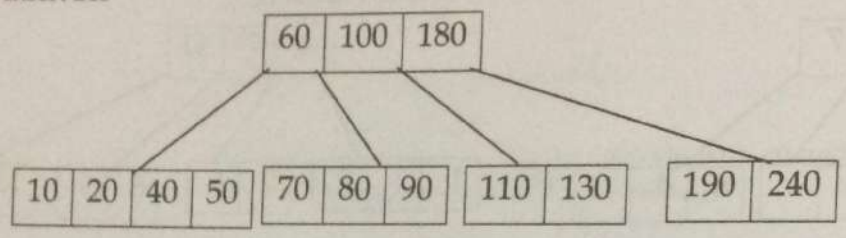
Step 4: Insert 100



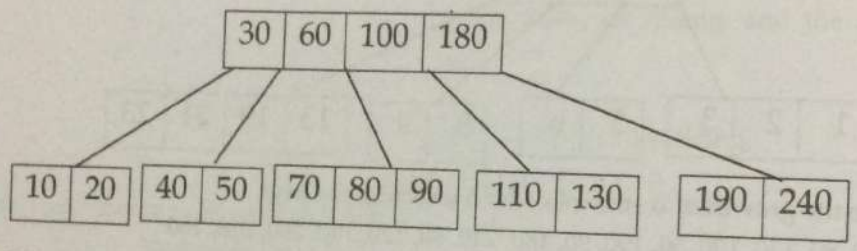
Step 5: Insert 50, 190, 90, 180



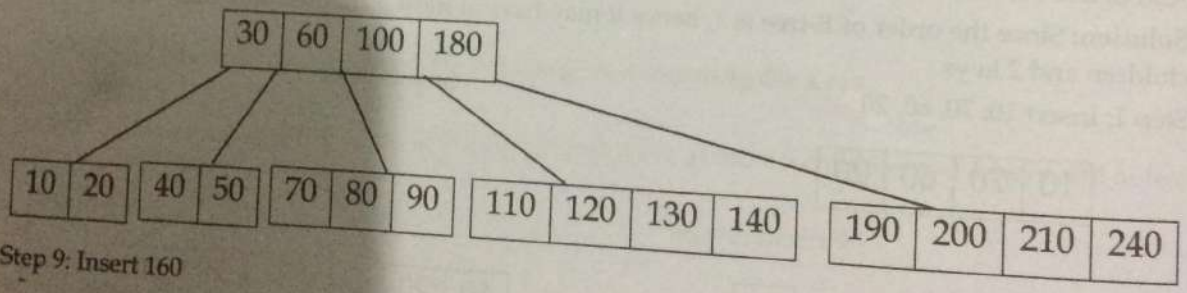
Step 6: Insert 240



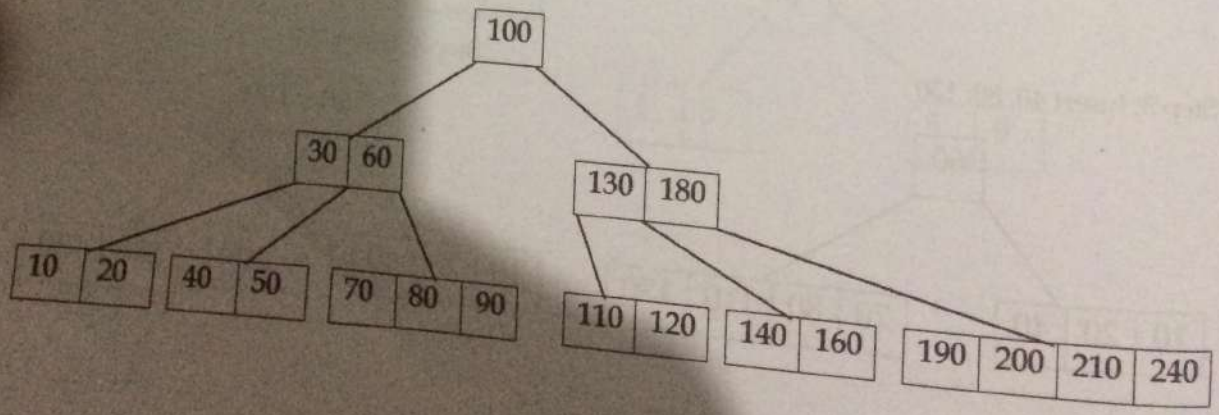
Step 7: Insert 30



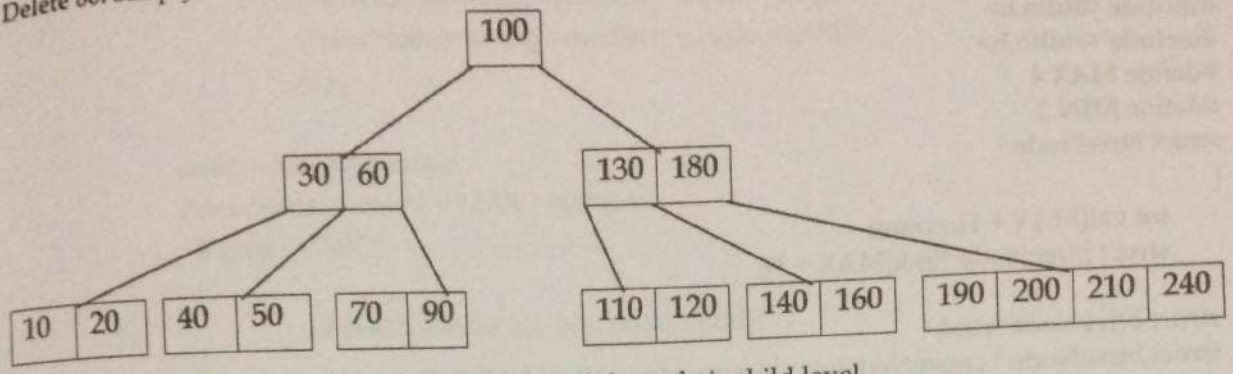
Step 8: Insert 120, 140, 200, 210



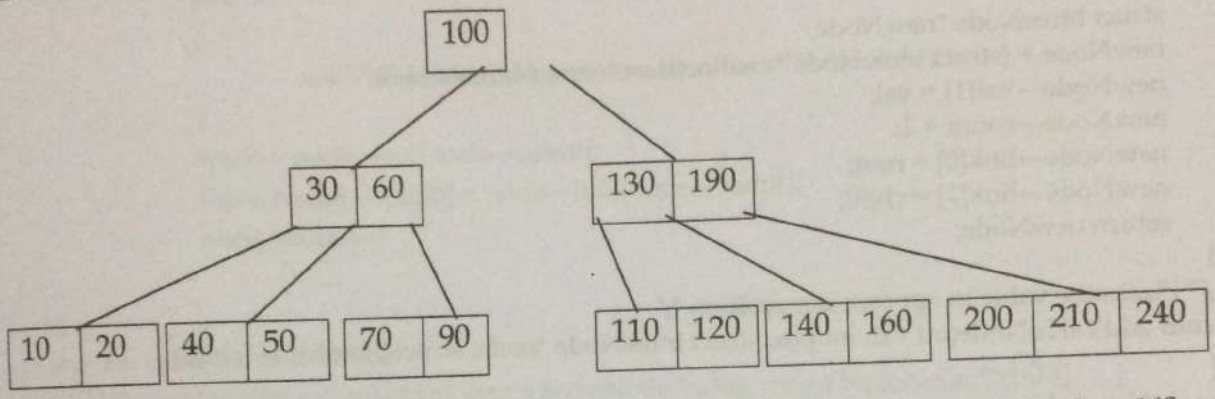
Step 9: Insert 160



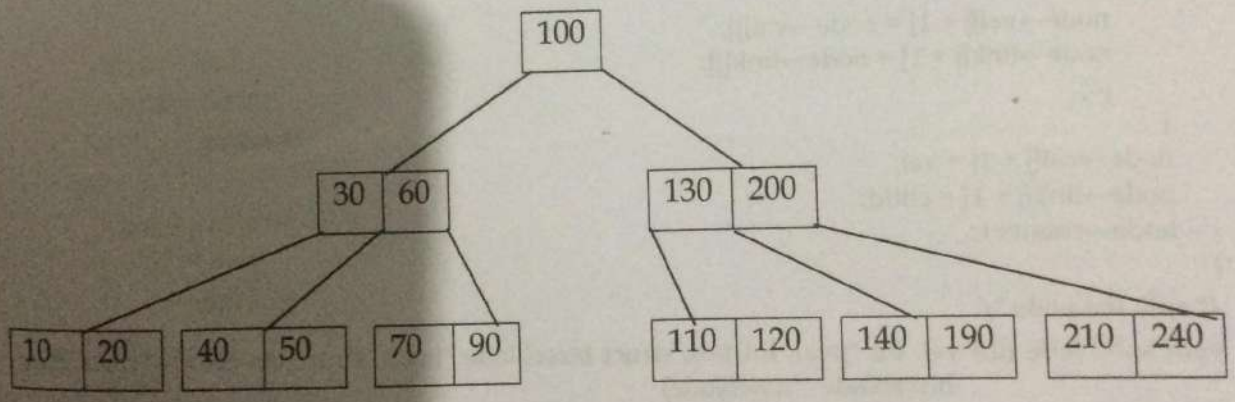
Delete 80: Simply delete 80 without performing any operation.



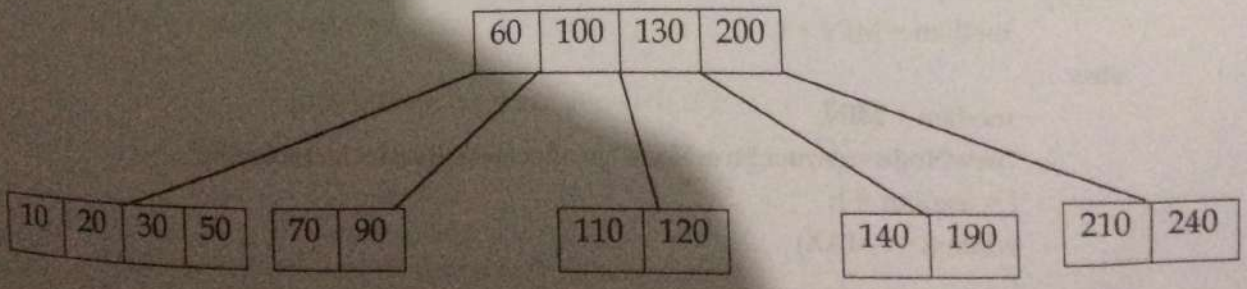
Delete 180: To delete 180 we need to promote 190 from their child level.



Delete 160: To delete 160 we need to demote 190 to their child note that contains key 160 and promote 200 from their child level.



Delete 40: To delete 40 we need to combine 10, 20, 30, 40 and 50 as a left child of 60 and also combine 60, 100, 130 and 200 to be root level.



Program in C for Insertion, Deletion and Traversal in B-Tree

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4
#define MIN 2
struct btreeNode
{
    int val[MAX + 1], count;
    struct btreeNode *link[MAX + 1];
};
struct btreeNode *root;
struct btreeNode * createNode(int val, struct btreeNode *child) /* creating new node */
{
    struct btreeNode *newNode;
    newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
    newNode→val[1] = val;
    newNode→count = 1;
    newNode→link[0] = root;
    newNode→link[1] = child;
    return newNode;
}
/* Places the value in appropriate position */
void addValToNode(int val, int pos, struct btreeNode *node, struct btreeNode *child)
{
    int j = node→count;
    while (j > pos)
    {
        node→val[j + 1] = node→val[j];
        node→link[j + 1] = node→link[j];
        j--;
    }
    node→val[j + 1] = val;
    node→link[j + 1] = child;
    node→count++;
}
/* split the node */
void splitNode (int val, int *pval, int pos, struct btreeNode *node, struct btreeNode *child, struct
                    btreeNode **newNode)
{
    int median, j;
    if (pos > MIN)
        median = MIN + 1;
    else
        median = MIN;
        *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
        j = median + 1;
        while (j <= MAX)
        {
```

```
                (*newNode)→val[j - median] = node→val[j];
                (*newNode)→link[j - median] = node→link[j];
                j++;
        }
        node→count = median;
        (*newNode)→count = MAX - median;
        if (pos <= MIN)
        {
                addValToNode(val, pos, node, child);
        }
        else
        {
                addValToNode(val, pos - median, *newNode, child);
        }
        *pval = node→val[node→count];
        (*newNode)→link[0] = node→link[node→count];
        node→count--;
}

/* sets the value val in the node */
int setValueInNode(int val, int *pval, struct btreeNode *node, struct btreeNode **child)    {
        int pos;
        if (!node)
        {
        *pval = val;
        *child = NULL;
                return 1;
        }
        if (val < node→val[1])
        {
                pos = 0;
        }
        else
        {
        for (pos = node→count;
                (val < node→val[pos] && pos > 1); pos--);
        if (val == node→val[pos])
                {
                        printf("Duplicates not allowed\n");
                return 0;
                }
        }
        if (setValueInNode(val, pval, node→link[pos], child))
```

```
        {
                if (node→count < MAX)
                {
                        addValToNode(*pval, pos, node, *child);
                }
                else {
                splitNode(*pval, pval, pos, node, *child, child);
                return 1;
                }
        }
        return 0;
}

/* insert value in B-Tree */
void insertion(int val)
{
    int flag, i;
    struct btreeNode *child;
    flag = setValueInNode(val, &i, root, &child);
    if (flag)
        root = createNode(i, child);
}

/* copy successor for the value to be deleted */
void copySuccessor(struct btreeNode *myNode, int pos)
{
    struct btreeNode *dummy;
    dummy = myNode→link[pos];
    for (;dummy→link[0] != NULL;)
        dummy = dummy→link[0];
    myNode→val[pos] = dummy→val[1];
}

/* removes the value from the given node and rearrange values */
void removeVal(struct btreeNode *myNode, int pos)
{
    int i = pos + 1;
        while (i <= myNode→count)
        {
                myNode→val[i - 1] = myNode→val[i];
                myNode→link[i - 1] = myNode→link[i];
                i++;
        }
```

```
        myNode→count--;
}

/* shifts value from parent to right child */
void doRightShift(struct btreeNode *myNode, int pos)
{
    struct btreeNode *x = myNode→link[pos];
    int j = x→count;
    while (j > 0)
    {
        x→val[j + 1] = x→val[j];
        x→link[j + 1] = x→link[j];
    }
    x→val[1] = myNode→val[pos];
    x→link[1] = x→link[0];
    x→count++;
    x = myNode→link[pos - 1];
    myNode→val[pos] = x→val[x→count];
    myNode→link[pos] = x→link[x→count];
    x→count--;
    return;
}

/* shifts value from parent to left child */
void doLeftShift(struct btreeNode *myNode, int pos)
{
    int j = 1;
    struct btreeNode *x = myNode→link[pos - 1];
    x→count++;
    x→val[x→count] = myNode→val[pos];
    x→link[x→count] = myNode→link[pos]→link[0];
    x = myNode→link[pos];
    myNode→val[pos] = x→val[1];
    x→link[0] = x→link[1];
    x→count--;
    while (j <= x→count)
    {
        x→val[j] = x→val[j + 1];
        x→link[j] = x→link[j + 1];
        j++;
    }
    return;
}
```

```
/* merge nodes */
void mergeNodes(struct btreeNode *myNode, int pos)
{
    int j = 1;
    struct btreeNode *x1 = myNode→link[pos];
    struct btreeNode *x2 = myNode→link[pos - 1];
    x2→count++;
    x2→val[x2→count] = myNode→val[pos];
    x2→link[x2→count] = myNode→link[0];
    while (j <= x1→count)
    {
        x2→count++;
        x2→val[x2→count] = x1→val[j];
        x2→link[x2→count] = x1→link[j];
        j++;
    }
    j = pos;
    while (j < myNode→count)
    {
        myNode→val[j] = myNode→val[j + 1];
        myNode→link[j] = myNode→link[j + 1];
        j++;
    }
    myNode→count–;
    free(x1);
}

/* adjusts the given node */
void adjustNode(struct btreeNode *myNode, int pos)
{
    if (!pos)
    {
        if (myNode→link[1]→count > MIN)
        {
            doLeftShift(myNode, 1);
        }
        else {
            mergeNodes(myNode, 1);
        }
    }
    else {
            if (myNode→count != pos)
```

```
        {
        if(myNode→link[pos - 1]→count > MIN)
                {
                        doRightShift(myNode, pos);
        }
                else {
                        if (myNode→link[pos + 1]→count > MIN)
                        {
                        doLeftShift(myNode, pos + 1);
                        }
                        else {
                        mergeNodes(myNode, pos);
                        }
        }
        }
        else
        {
        if (myNode→link[pos - 1]→count > MIN)
                        doRightShift(myNode, pos);
        else
                        mergeNodes(myNode, pos);
        }
    }
}

/* delete val from the node */
int delValFromNode(int val, struct btreeNode *myNode)
{
    int pos, flag = 0;
    if (myNode)
    {
        if (val < myNode→val[1])
        {
            pos = 0;
            flag = 0;
        }
        else {
                for (pos = myNode→count;
                        (val < myNode→val[pos] && pos > 1); pos--);
                if (val == myNode→val[pos]) {
                        flag = 1;
            } else {
                flag = 0;
```

```
            }
        }
        if (flag)
          {
            if (myNode→link[pos - 1])
                {
                  copySuccessor(myNode, pos);
                  flag = delValFromNode(myNode→val[pos], myNode→link[pos]);
                  if (flag == 0) {
                        printf("Given data is not present in B-Tree\n");
                  }
            }
                else {
                  removeVal(myNode, pos);
            }
        }
         else {
            flag = delValFromNode(val, myNode→link[pos]);
        }
        if (myNode→link[pos])
          {
            if (myNode→link[pos]→count < MIN)
                        adjustNode(myNode, pos);
          }
    }
    return flag;
}

/* delete value from B-tree */
void deletion(int val, struct btreeNode *myNode)
{
    struct btreeNode *tmp;
    if (!delValFromNode(val, myNode))
    {
        printf("Given value is not present in B-Tree\n");
        return;
    }
    else
    {
            if (myNode→count == 0)
            {
            tmp = myNode;
            myNode = myNode→link[0];
```

```
            free(tmp);
            }
        }
        root = myNode;
        return;
}

/* search val in B-Tree */
void searching(int val, int *pos, struct btreeNode *myNode)
{
    if (!myNode)
    {
        return;
    }
    if (val < myNode→val[1])
    {
        *pos = 0;
    }
    else {
            for (*pos = myNode→count;
            (val < myNode→val[*pos] && *pos > 1); (*pos)--);
            if (val == myNode→val[*pos]) {
            printf("Given data %d is present in B-Tree", val);
            return;
        }
    }
    searching(val, pos, myNode→link[*pos]);
    return;
}

/* B-Tree Traversal */
void traversal(struct btreeNode *myNode)
{
    int i;
    if (myNode)
    {
        for (i = 0; i < myNode→count; i++)
        {
            traversal(myNode→link[i]);
            printf("%d ", myNode→val[i + 1]);
        }
        traversal(myNode→link[i]);
    }
```

```
}

int main()
{
    int val, ch;
    while (1)
    {
        printf("1. Insertion\t2. Deletion\n");
        printf("3. Searching\t4. Traversal\n");
        printf("5. Exit\n Enter your choice :");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("Enter your input:");
                scanf("%d", &val);
                insertion(val);
                break;
            case 2:
                printf("Enter the element to delete:");
                scanf("%d", &val);
                deletion(val, root);
                break;
            case 3:
                printf("Enter the element to search:");
                scanf("%d", &val);
                searching(val, &ch, root);
                break;
            case 4:
                traversal(root);
                break;
            case 5:
                exit(0);
            default:
                printf("U have entered wrong option!!\n");
                break;
        }
        printf("\n");
    }
}
```

Output

1. Insertion 2. Deletion

```
3. Searching 4. Traversal
5. Exit
Enter your choice:1
Enter your input:70

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:1
Enter your input:17

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:1
Enter your input:67

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:1
Enter your input:89

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:4
17 67 70 89

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:3
Enter the element to search:70
Given data 70 is present in B-Tree

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:2
Enter the element to delete:17

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:4
67 70 89

1. Insertion 2. Deletion
3. Searching 4. Traversal
```

5. Exit
Enter your choice: 5

B+ Trees

B+ tree is a data structure often used in the implementation of database indexes. Each node of the tree contains an ordered list of keys and pointers to lower level nodes in the tree. These pointers can be thought of as being between each of the keys. To search for or insert an element into the tree, one loads up the root node, finds the adjacent keys that the searched-for value is between, and follows the corresponding pointer to the next node in the tree.

B+ trees use some clever balancing techniques to make sure that all of the leaves are always on the same level of the tree, that each node is always at least half full of keys, therefore the height of the tree is always at most ceiling(log(k)) of base ceiling(n/2) where k is the number of values in the tree and n is the maximum number of pointers = (maximum number of nodes + 1) in each block. This means that only a small number of pointer traversal is necessary to search for a value if the number of keys in a node is large. This is crucial in a database because the B+ tree is on disk.

B+ trees can also be used outside of the disk, but generally a balanced binary search tree or a skip list or something should provide better performance in memory, where pointer following is no more expensive than finding the right pointer to follow. B trees, B+ trees comes handy when we have huge which can't fit in main memory and has to perform disk seek operations. These two self balancing trees will reduces number of disk seeks.

The B+ Tree structure

The tree has a single node at the top, called the root node. The root node points to two or more blocks, called child nodes. Each child nodes points to further child nodes and so on. The B+-Tree consists of two types of (1) internal nodes and (2) leaf nodes

- Internal nodes point to other nodes in the tree.
- Leaf nodes point to data in the database using data pointers. Leaf nodes also contain an additional pointer, called the sibling pointer, which is used to improve the efficiency of certain types of search.
- All the nodes in a B+ Tree must be at least half full except the root node which may contain a minimum of two entries. The algorithms that allow data to be inserted into and deleted from a B+ Tree guarantee that each node in the tree will be at least half full.
- Searching for a value in the B+ Tree always starts at the root node and moves downwards until it reaches a leaf node.
- Both internal and leaf nodes contain key values that are used to guide the search for entries in the index.
- The B+ Tree is called a balanced tree because every path from the root node to a leaf node is the same length. A balanced tree means that all searches for individual values require the same number of nodes to be read from the disc.
- An **internal node** in a B+ Tree consists of a set of key values and pointers. The set of keys and values are ordered so that a pointer is followed by a key value. The last key value is followed by one pointer.
- Each pointer points to nodes containing values that are less than or equal to the value of the key immediately to its right.
- The last pointer in an internal node is called the **infinity pointer**. The infinity pointer points to a node containing key values that are greater than the last key value in the node.

- When an internal node is searched for a key value, the search begins at the leftmost key value and moves rightwards along the keys.

Properties of a B+ Tree

1. The root node points to at least two nodes.
2. All non-root nodes are at least half full.
3. For a tree of order m, all internal nodes have m-1 keys and m pointers.
4. A B+-Tree grows upwards.
5. A B+-Tree is balanced.
6. Sibling pointers allow sequential searching.

B+ Tree Data Insertion

Algorithm

1. Start
2. If the bucket is not full (at most b 1 entries after the insertion), add the record
3. Otherwise, split the bucket.
    1. Allocate new leaf and move half the buckets elements to the new bucket.
    2. Insert the new leafs smallest key and address into the parent.
    3. If the parent is full, split it too.
        - Add the middle key to the parent node.
    4. Repeat until a parent is found that need not split.
4. If the root splits, create a new root which has one key and two pointers. (i.e. the value that gets pushed to the new root gets removed from the original node)
5. Stop

B+ Tree Data Deletion

Algorithm

1. Start
2. Start at the root and go up to leaf node containing the key K
3. Find the node n on the path from the root to the leaf node containing K

    A. If n is root, remove K

        a. if root has more than one keys, done

        b. if root has only K

            i. if any of its child node can lend a node

                Borrow key from the child and adjust child links

            ii. Otherwise merge the children nodes it will be new root

        c. If n is a internal node, remove K

            i) If n has at least ceil(m/2) keys, done!

            ii) If n has less than ceil(m/2) keys,

                If a sibling can lend a key,

                Borrow key from the sibling and adjust keys in n and the parent node

                Adjust child links

                Else

                Merge n with its sibling

Adjust child links

d. If n is a leaf node, remove K

i) If n has at least ceil(M/2) elements, done!

In case the smallest key is deleted, push up the next key

ii) If n has less than ceil(m/2) elements

If the sibling can lend a key

Borrow key from a sibling and adjust keys in n and its parent node

Else

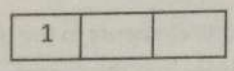
Merge n and its sibling

Adjust keys in the parent node

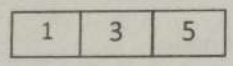
4. Stop

Example: Insert 1, 3, 5, 7, 9, 2, 4, 6, 8 to B+ tree of order 3

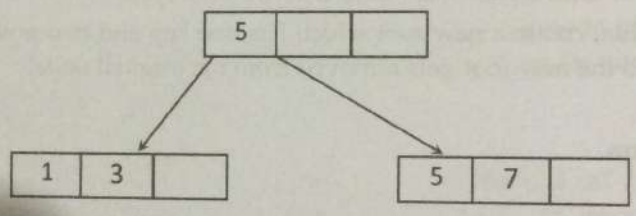
Insert 1:



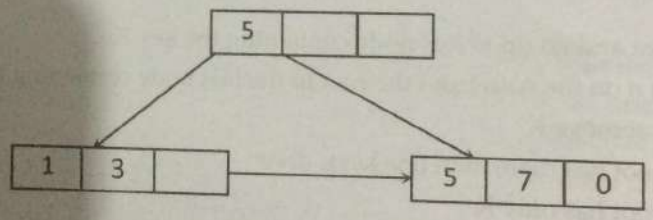
Insert 3, 5:



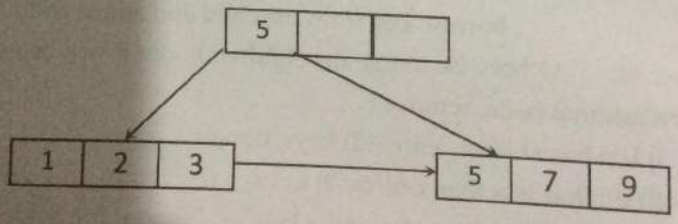
Insert 7:



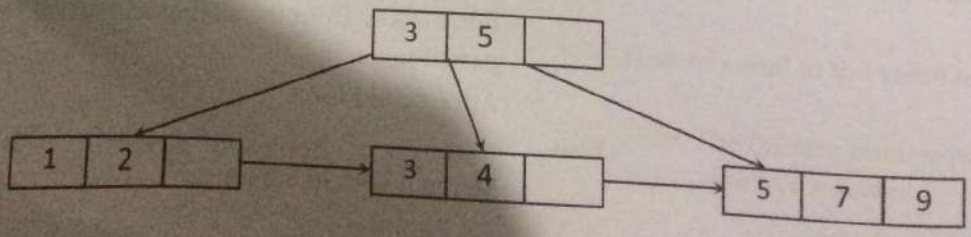
Insert 9:



Insert 2:

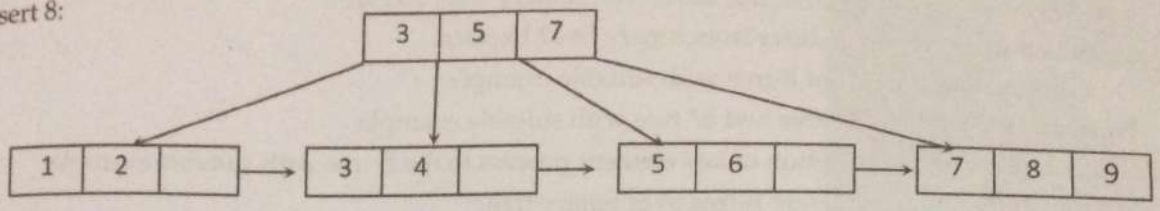


Insert 4:



Insert 6:

Insert 8:



Differences between a B tree and a B+ tree

B+ Trees are different from B Trees with following two properties:

1. B+ trees don't store data pointer in interior nodes, they are only stored in leaf nodes. This is not optional as in B-Tree. This means that interior nodes can fit more keys on block of memory.
2. The leaf nodes of B+ trees are linked, so doing a linear scan of all keys will requires just one pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree. This property can be utilized for efficient search as well since data is stored only in leafs.

B* Trees

In computer science, B* pronounced "B star" is a best-first graph search algorithm that finds the least-cost path from a given initial node to any goal node (out of one or more possible goals). First published by Hans Berliner in 1979, it is related to the A* search algorithm. The algorithm stores intervals for nodes of the tree as opposed to single point-valued estimates. Then, leaf nodes of the tree can be searched until one of the top level nodes has an interval which is clearly "best." Leaf nodes of a B*-tree are given evaluations that are intervals rather than single numbers. The interval is supposed to contain the true value of that node. If all intervals attached to leaf nodes satisfy this property, then B* will identify an optimal path to the goal state.

B* systematically expands nodes in order to create "separation," which occurs when the lower bound of a direct child of the root is at least as large as the upper bound of any other direct child of the root. A tree that creates separation at the root contains a proof that the best child is at least as good as any other child.

Applications of B* tree

Andrew Palay applied B* to chess. Endpoint evaluations were assigned by performing null-move searches. There is no report of how well this system performed compared to alpha-beta pruning search engines running on the same hardware.

The Maven (Scrabble) program applied B* search to endgames. Endpoint evaluations were assigned using a heuristic planning system. This program succeeded splendidly, establishing the gold standard for endgame analysis. The B* search algorithm has been used to compute optimal strategy in a sum game of a set of combinatorial games.

Exercise

1. What is multi-way tree? How it is differ from binary tree? Explain.
2. What is B-tree? How it is differ from binary tree? Explain.
3. Describe various variant of B-tree with suitable example.
4. Differentiate between B+ tree and B* tree with suitable example.
5. Explain insertion and deletion of key element process to the B-tree with suitable example.
6. What are advantages of using B-tree over binary tree?
7. Is B-tree is family of binary tree? Explain.
8. What are advantages and disadvantages of B-tress?
9. Describe application areas of B-tree with suitable example
10. Why are B-trees better than inverted files?
11. Write complete Java program to show all basic operations of B-tree.
12. Construct B-Tree of order 4 by using following data items:
    [5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8] and delete data items [2, 21, 10, 3, 4] from given B-tree.
13. Suppose that you have an application in which you want to use B-trees. Suppose that the computer you will be using has disk blocks holding 4096 bytes, the key is 4 bytes long, each child pointer (which is a disk block id) is 4 bytes, the parent is 4 bytes long and the data record reference (which is a disk block id along with a offset within the block) is 8 bytes. You have an application in which you want to store 1,000,000 items in your B-tree. What value would you select for t? (Show how you derived it.) What is the maximum number of disk pages that will be brought into main memory during a search? Remember that the root is kept in main memory at all times.
14. In this problem, assume that every B tree node has a reference to its parent (except for the root node) as well as its children (except for leaves), so that it is easy to "move up or down" the tree. Suppose you have a direct reference to the leaf holding a data item with a key k. a) Describe in English an algorithm for finding the predecessor of k in the B tree (or determining that k is the minimum element and therefore has no predecessor). b) Give the worst-case number of total nodes accessed by your algorithm in terms of the tree height h. Describe briefly a worst-case situation. c) Give the best-case number of total nodes accessed by your algorithm. Explain briefly why most keys would exhibit the best-case behavior.
15. Consider a B+-tree in which the maximum number of keys in a node is 5. What is the minimum number of keys in any non-root node?
16. For 8 keys and 6 slots in a hashing table with uniform hashing and chaining, what is the expected number of items that hash to a particular location?
17. Suppose that a non-leaf node in a B-tree contains 42 entries. How many children does the node have?
18. Draw an example of a B-tree with four nodes and seven integer entries. The value of minimum is 1 for this tree.
19. Suppose that 'a' and b are two positive integers and n is some non-negative number. Write an equation to show the relationship between log base a of n and log base b of n. Give a derivation to show that the relationship is valid.
20. List the rules for a B-tree and determine whether a tree satisfies these rules.

•••

Chapter

9

Sorting and Searching

Introduction

Sorting is the process of ordering elements in an array in specific order e.g. ascending or descending on the basis of value, chronological ordering a records, priority order.

Sorting is categorized as internal sorting and external sorting.

1. **Internal sorting:** By internal sorting means we are arranging the numbers within the array only which is in computer primary memory.
2. **External sorting:** External sorting is the sorting of numbers from the external file by reading it from secondary memory.

Let P be a list of n elements $P_1, P_2, P_3, \ldots\ldots\ldots\ldots, P_n$ in memory. Sorting P means arranging the contents of P in either increasing or decreasing order i.e. $P_1 \leq P_2 \leq P_3 \leq P_4 \leq P_5 \ldots\ldots\ldots\ldots \leq P_n$

There are n elements in the list, therefore there are n! Ways to arrange them.

Consider a list of values: 2 ,4 ,6 ,8 ,9 ,1 ,22 ,4 ,77 ,8 ,9

After sorting the values: 1, 2, 4, 4, 6, 8, 8, 9, 9, 22, 77

In-place

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list. For example, Insertion Sort and Selection Sorts are in-place sorting algorithms as they do not use any additional space for sorting the list and a typical implementation of Merge Sort is not in-place, also the implementation for counting sort is not in-place sorting algorithm.

Stable

A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key remain sorted on the first key.

Why we using sorting?

We know that searching a sorted array is much easier than searching an unsorted array. This is especially true for people. That is, finding a person's name in a phone book is easy, but finding a phone number without knowing the person's name is virtually impossible. As a result, any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted. The following are some more examples.

- Words in a dictionary are sorted
- Files in a directory are often listed in sorted order.
- The index of a book is sorted
- The card catalog in a library is sorted by both author and title.
- A listing of course offerings at a university is sorted, first by department and then by course number.
- Many banks provide statements that list checks in increasing order by check number.
- In a news paper, the calendar of events in a schedule is generally sorted by date.
- Musical compact disks in a record store are generally sorted by recording artist.
- In the programs printed for graduation ceremonies, departments are listed in sorted order and then students in those departments are listed in sorted order.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. The basic idea of this sort is to pass through the array sequentially several times. Each pass consists of comparing each element in the array with its successor (for example a[i] with a[i + 1]) and interchanging the two elements if they are not in the proper order. For example, consider the following array:

Characteristics of Bubble Sort:

- Large values are always sorted first.
- It only takes one iteration to detect that a collection is already sorted.
- The best time complexity for Bubble Sort is O(n). The average and worst time complexity is $O(n^2)$.
- The space complexity for Bubble Sort is O(1), because only single additional memory space is required.

Tracing: Sort the following data items by using Bubble sort

A[ ]={25, 57, 48, 37, 12, 92, 86, 33}

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass 1	25	48	37	12	57	86	33	92
Pass 2	24	37	12	48	57	33	86	92
Pass 3	24	12	37	48	33	57	86	92
Pass 4	12	24	37	33	48	57	86	92
Pass 5	12	24	33	37	48	57	86	92
Pass 6	12	24	33	37	48	57	86	92
Pass 7	12	24	33	37	48	57	86	92
Pass 8	12	24	33	37	48	57	86	92

Algorithm

Let's look at implementing the optimized version of bubble sort:

1. Start
2. For the first iteration, compare all the elements (n). For the subsequent runs, compare (n-1) (n-2) and so on.
3. Compare each element with its right side neighbor.
4. Swap the smallest element to the left.
5. Keep repeating steps 1-3 until the whole list is covered.
6. Stop

Pseudo code

```
BubbleSort (A, n)
{
        for(i = 0; i<n-1; i++)
        {
                for(j = 0; j < n-i-1; j++)
                {
                        if(A[j] > A[j+1])
                        {
                                temp = A[j];
                                A[j] = A[j+1];
                                A[j+1] = temp;
                        }
                }
        }
}
```

Time Complexity

Inner loop executes (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ... +2 +1

= n (n-1)/2

$= O(n^2)$

There is no best-case linear time complexity for this algorithm.

Complete program in C for Bubble sort

```
#include<stdio.h>
#include<conio.h>
void bubble(int[ ],int);
void main( )
{
        int n;
        int a[100], i;
        printf("Enter no of data items:\n");
        scanf("%d", &n);
        printf("Enter %d data items:\n", n);
        for(i=0; i<n; i++)
                scanf("%d", &a[i]);
         printf("The data items before sorting:\n");
         for(i=0; i<n; i++)
                printf("%d\t", a[i]);
         bubble(a, n);
         printf("The data items after sorting:\n");
         for(i=0; i<n; i++)
                printf("%d\t", a[i]);
}
void bubble(int a[ ], int n)  /*bubble function*/
{
        int i, j, temp;
        for(i=0; i<n-1; i++)
        {
         for(j=0; j<n-i-1; j++)
         {
                temp=a[j];
                if(a[j]>a[j+1])
                {
                        a[j]=a[j+1];
                        a[j+1]=temp;
                }
         }
    }
}
```

Input/output

Enter number of data points

10
Enter 10 numbers
66 5 44 3 55 44 2 1 7 9
The data items before sorting:
66 5 44 3 55 44 2 1 7 9
The data items after sorting:
1 2 3 5 7 9 44 44 55 66

Selection Sort

Selection Sort is about picking/selecting the smallest element from the list and placing it in the sorted portion of the list. Initially, the first element is considered the minimum and compared with other elements. During these comparisons, if a smaller element is found then that is considered the new minimum. After completion of one full round, the smallest element found is swapped with the first element. This process continues till all the elements are sorted.

Idea: Find the least (or greatest) value in the array, swap it into the leftmost (or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly. Let a[n] be a linear array of n elements. The selection sort works as follows:

Pass 1: Find the location loc of the smallest element from the list of n elements a[0], a[1], a[2], a[3], ...,a[n-1] and then interchange a[loc] and a[0].

Pass 2: Find the location loc of the smallest element from the sub-list of n-1 elements a[1], a[2], a[3], ...,a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted

.....

.....and so on.

Then we will get the sorted list a[0]<=a[1]<= a[2]<=a[3]<= .....<=a[n-1].

Tracing: Sort the following data items by using Selection sort

A[ ]={25, 57, 48, 37, 12, 92, 86, 33}

Solution:

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass 1	12	57	48	37	25	92	86	33
Pass 2	12	25	48	37	57	92	86	33
Pass 3	12	25	33	37	57	92	86	48
Pass 4	12	25	33	37	57	92	86	48
Pass 5	12	25	33	37	48	92	86	57
Pass 6	12	25	33	37	48	57	86	92
Pass 7	12	25	33	37	48	57	86	92
Pass 8	12	25	33	37	48	57	86	92

Algorithm

1. Start
2. Consider the first element to be sorted and the rest to be unsorted
3. Assume the first element to be the smallest element.

4. Check if the first element is smaller than each of the other elements:
   - If yes, do nothing
   - If no, choose the other smaller element as minimum and repeat step 3
5. After completion of one iteration through the list, swap the smallest element with the first element of the list.
6. Now consider the second element in the list to be the smallest and so on till all the elements in the list are covered.
7. Stop

Pseudo code

```
SelectionSort(A)
{
        for( i = 0; i< n ; i++)
        {
                least =A[i];
                p=i;
                for ( j = i + 1; j < n ; j++)
                {
                        if (A[j] < A[i])
                        {
                                least = A[j];
                                p=j;
                        }
                }
                swap (A[i], A[p]);
        }
}
```

Time Complexity

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ... +2 +1

$= O(n^2)$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Complete program in C for Selection sort

```
#include<stdio.h>
#include<conio.h>
void selection(int[ ], int);
void main( )
{
        int n;
        int a[100], i;
        printf("Enter no of data items:\n");
```

```
        scanf("%d", &n);
        printf("Enter %d data items:\n", n);
        for(i=0; i<n; i++)
                scanf("%d", &a[i]);
        printf("The data items before sorting:\n");
        for(i=0; i<n; i++)
                printf("%d\t", a[i]);
        selection(a, n);
        printf("The data items after sorting:\n");
        for(i=0; i<n; i++)
                printf("%d\t", a[i]);
        getch( );
}
/*selection Function*/
void selection(int a[ ], int n)
{
        int i, j, temp, index, least;
        for(i=0;i<n;i++)
        {
                least=a[i];
                index=i;
                for(j=i+1;j<n;j++)
                {
                if(a[j]<least)
                {
                                least=a[j];
                                index=j;
                        }
                }
                if(i!=index)
                {
                        temp=a[i];
                        a[i]=a[index];
                        a[index]=temp;
                }
        }
}
```

Input/output

Enter number of data points

10

Enter 10 numbers

66 5 44 3 55 44 2 1 7 9

The data items before sorting:

66 5 44 3 55 44 2 1 7 9

The data items after sorting:

1 2 3 5 7 9 44 44 55 66

Insertion Sort

As the name suggests, in Insertion Sort, an element gets compared and inserted into the correct position in the list. To apply this sort, you must consider one part of the list to be sorted and the other to be unsorted. To begin, consider the first element to be the sorted portion and the other elements of the list to be unsorted. Now compare each element from the unsorted portion with the element in the sorted portion. Then insert it in the correct position in the sorted part. Insertion Sort works best with small number of elements. The worst case runtime complexity of Insertion Sort is $O(n^2)$ similar to that of Bubble Sort. However, Insertion Sort is considered better than Bubble sort.

Idea: Like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are finally sorted.

Suppose an array a[n] with n elements. The insertion sort works as follows:

Pass 1: a[0] by itself is trivially sorted.

Pass 2: a[1] is inserted either before or after a[0] so that a[0], a[1] is sorted.

Pass 3: a[2] is inserted into its proper place in a[0],a[1] that is before a[0], between a[0] and a[1], or after a[1] so that a[0],a[1],a[2] is sorted.

.....

Pass n: a[n-1] is inserted into its proper place in a[0],a[1],a[2],.....,a[n-2] so that a[0],a[1],a[2],.....,a[n-1] is sorted with n elements.

Tracing: Sort the following data items by using Insertion sort

A[ ]={25, 57, 48, 37, 12, 92, 86, 33}

Solution:

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
After a[0..1] is sorted (pass 1)	25	57	48	37	12	92	86	33
After a[0..2] is sorted (pass 2)	25	57	48	37	12	92	86	33
After a[0..3] is sorted (pass 3)	25	48	57	37	12	92	86	33
After a[0..4] is sorted (pass 4)	25	37	48	57	12	92	86	33
After a[0..5] is sorted (pass 5)	12	25	37	48	57	92	86	33
After a[0..6] is sorted (pass 6)	12	25	37	48	57	92	86	33
After a[0..7] is sorted (pass 7)	12	25	37	48	57	86	92	33
After a[0..8] is sorted (pass 8)	12	25	33	37	48	57	86	92

Algorithm

1. Start
2. Consider the first element to be sorted and the rest to be unsorted
3. Compare with the second element:
    1. If the second element< the first element, insert the element in the correct position of the sorted portion
    2. Else, leave it as it is
4. Repeat 1 and 2 until all elements are sorted
5. Stop

Pseudo code

```
Insertion( A, n)
{
        for i =1 to n
        {
                temp =A[i]
                j = i-1
                while( j>= 0 && A[j] > temp)
                {
                        A[j+1] = A[j]
                        j = j - 1
                }
        }
        A[j+1] = temp
}
```

Complete program in C for Insertion sort

```
#include<stdio.h>
#include<conio.h>
void insertion(int[ ], int);
void main( )
{
        int n;
        int a[100], i;
        clrscr( );
        printf("Enter no of data items:\n");
        scanf("%d", &n);
        printf("Enter %d data items:\n", n);
        for(i=0; i<n; i++)
        {
                scanf("%d", &a[i]);
        }
```

```
        printf("The data items before sorting:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
        insertion(a, n);
        printf("The data items after sorting:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
        getch( );
}
/*insertion Function*/
void insertion(int a[ ], int n)
{
        int i, j, temp;
        for(i=0; i<n; i++)
        {
                temp = a[i];
                j=i-1;
                while((temp<a[j])&&j>=0)
                {
                    a[j+1]=a[j];
                    j--;
                }
                a[j+1]=temp;
        }
}
```

Input/output

Enter number of data points
10
Enter 10 numbers
69 5 44 3 55 45 2 1 7 9
The data items before sorting:
69 5 44 3 55 45 2 1 7 9
The data items after sorting:
1 2 3 5 7 9 44 45 55 69

Divide-and-conquer algorithms

An important problem-solving technique that makes use of recursion is divide and conquer. A divide-and-conquer algorithm is an efficient recursive algorithm that consists of two parts:

- **Divide,** in which smaller problems are solved recursively (except, ofcourse, base cases)

- **Conquer,** in which the solution to the original problem is then formed from the solutions to the sub-problems

Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer algorithms, whereas routines whose text contains only one recursive call are not. Consequently, the recursive routines presented so far in this section are not divide-and-conquer algorithms. Also, the sub-problems usually must be disjoint (i.e., essentially no overlapping), so as to avoid the excessive costs seen in the sample recursive computation of the Fibonacci numbers.

Quick Sort

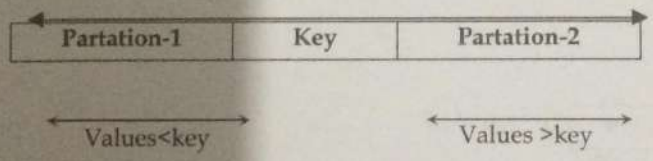
As its name implies, **quick sort** is a fast divide-and-conquer algorithm. Its average running time is O(n logn) . Its speed is mainly due to a very tight and highly optimized inner loop. It has quadratic worst-case performance, which can be made statistically unlikely to occur with a little effort. On the one hand, the quicksort algorithm is relatively simple to understand and prove correct because it relies on recursion. On the other hand, it is a tricky algorithm to implement because minute changes in the code can make significant differences in running time.

Quick sort is a very efficient sorting algorithm invented by C.A.R. Hoare. It has two phases:

- The partition phase and
- The sort phase

As we will see, most of the work is done in the partition phase - it works out where to divide the work. The sort phase simply sorts the two smaller problems that are generated in the partition phase. This makes Quick sort a good example of the **divide and conquers** strategy for solving problems.

- **Divide**
  Partition the array A[l…r] into two sub-arrays A[l…pivot] and A[pivot+1...r], such that each element of A[l...pivot] is smaller than or equal to each element in A[pivot+1...r].
- **Conquer**
  Recursively sort A[l…pivot] and A[pivot+1…r] using Quick sort
- **Combine**
  Trivial: the arrays are sorted in place. No additional work is required to combine them.



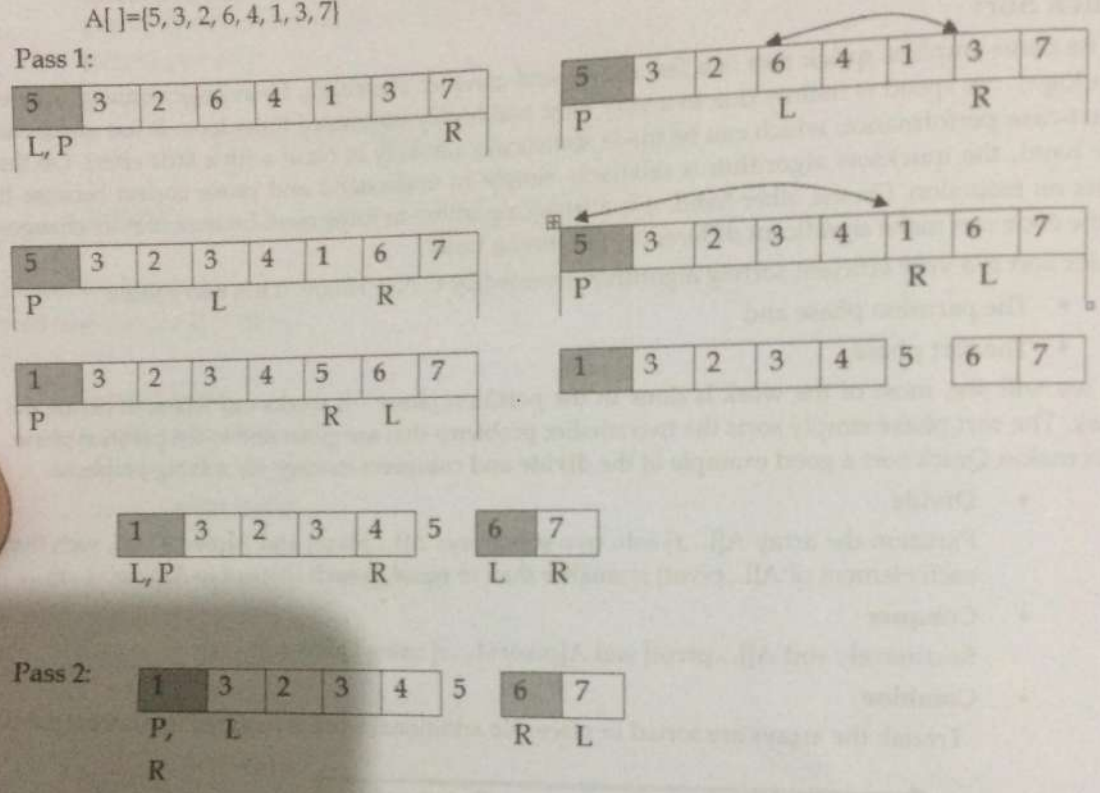
Quick Sort is quite different and complicated from the sorting techniques we have seen so far. In this technique, we select the first element and call it the pivot. The idea is to group elements such that elements to the left of the pivot are smaller than the pivot and the elements to the right of the pivot are larger than the pivot. This is done by maintaining two pointers left and right. The left pointer points to the first element after the pivot. Let us refer to the element pointed by left as **element**. Similarly, the right pointer points to the farthest element on the right side of the list. Let us call this element as **element.** At each step, compare element with pivot and element with pivot. Remember **element**< pivot and element > pivot. If these conditions are not satisfied, the element and element are swapped. Else, left pointer is incremented by 1 and the right pointer is decremented by 1. When left ≥ right, the pivot is swapped with either the element or element. The pivot element will be in its correct position. Then continue the quick sort on the left half and the right half of the list.

Quick Sort works best with small and large number of elements. The worst case runtime complexity of Quick Sort is O(n2) similar to that of Insertion and Bubble Sort but it can be improved to O(n log(n)) as discussed in the previous section. Unlike Merge Sort this doesn't have the disadvantage of using extra memory or space. That is why this is one of the most commonly used sorting techniques.

Tracing: Sort the following data items by using Quick sort

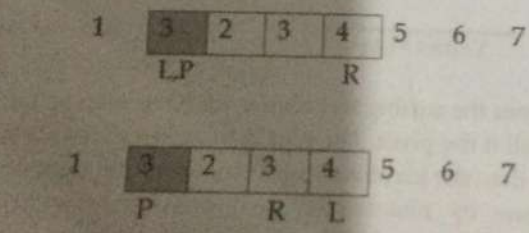
A[ ]={5, 3, 2, 6, 4, 1, 3, 7}

Pass 1:

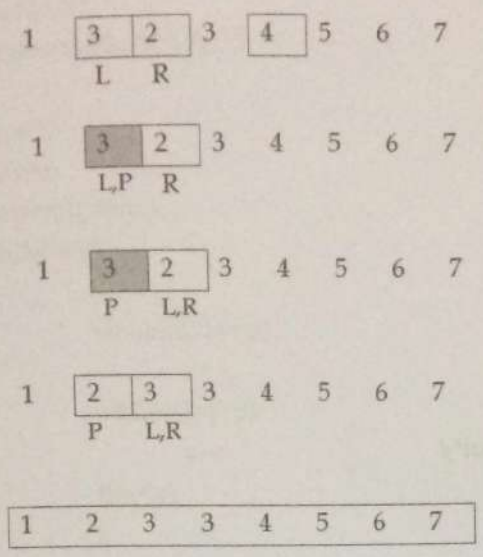


Pass 2:

Pass 3:



Pass 4:



Algorithm

1. Start
2. Choose a pivot
3. Set a left pointer and right pointer
4. Compare the left pointer element (lelement) with the pivot and the right pointer element (relement) with the pivot.
5. Check if lelement<pivot and relement>pivot:
    1. If yes, increment the left pointer and decrement the right pointer
    2. If not, swap the lelement and relement
6. When left >= right, swap the pivot with either left or right pointer.
7. Repeat steps 1 - 5 on the left half and the right half of the list till the entire list is sorted.
8. Stop

Pseudo code

```
QuickSort(A, l, r)
{
        if(l<r)
        {
                p = Partition(A, l, r);
                QuickSort(A, l, p-1);
                QuickSort(A, p+1, r);
        }
}
Partition(A,l,r)
{
        x =l;
        y =r ;
        p = A[l];
```

```
        while(x<y)
        {
                while(A[x] <= p)
                        x++;
                while(A[y] >=p)
                        y--;
                if(x<y)
                        swap(A[x],A[y]);
        }
        A[l] = A[y];
        A[y] = p;
        return y;           /*return position of pivot*/
}
```

Time Complexity

Best Case

Quick sort gives best time complexity when elements are divided into two partitions of equal size; therefore recurrence relation for this case is;

$$T(n) = 2T(n/2) + O(n)$$

By solving this recurrence, we get,

$$T(n)=O(n \log n)$$

Worst case

Quick sort gives worst case when elements are already sorted. In this case one partition contains the n-1 elements and another partition contains no element. Therefore, its recurrence relation is;

$$T(n) = T(n-1) + O(n)$$

By solving this recurrence relation, we get

$$T(n)=O(n^2)$$

Average case

It is the case between best case and worst case. All permutations of the input numbers are equally likely. On a random input array, we will have a mix of well balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree. Suppose we are alternate: Balanced, Unbalanced, Balanced.

$B(n)= 2UB(n/2) + \Theta(n)$ Balanced

$UB(n)= B(n-1) + \Theta(n)$ Unbalanced

Solving: $B(n)= 2(B(n/2-1) + \Theta(n/2)) + \Theta(n)$

$= 2B(n/2-1) + \Theta(n)$

$= \Theta(n \log n)$

Complete program in C for Quick sort

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

```
#include<conio.h>
int partition(int a[10], int l, int r)
{
        int x=l;
        int y=r;
        int p=a[l], temp;
        while(x<y)
        {
                while(a[x]<=p)
                        x++;
                while(a[y]>p)
                        y--;
                if(x<y)
                {
                        temp=a[x];
                        a[x]=a[y];
                        a[y]=temp;
                }
        }
        a[l]=a[y];
        a[y]=p;
        return y;
}
void quick(int a[10], int l, int r)
{
        int p;
        if(l<r)
        {
                p=partition(a, l, r);
                quick(a, l, p-1);
                quick(a, p+1, r);
        }
}
void main( )
{
        int a[100], n, i, l, r;
        printf("Enter no of elements\n");
        scanf("%d", &n);
        printf("Enter %d elements", n);
        l=0;
        r=n-1;
        for(i=0; i<n; i++)
```

```
        {
                scanf("%d", &a[i]);
        }
        printf("elements before sort:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
        quick(a, l, r);
        printf("elements after sort:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
        getch( );
}
```

Input/output

Enter number of data points
10
Enter 10 numbers
69 5 44 3 55 45 2 1 7 9
The data items before sorting:
69 5 44 3 55 45 2 1 7 9
The data items after sorting:
1 2 3 5 7 9 44 45 55 69

Time Complexity

Best Case

Quick sort gives best time complexity when elements are divided into two partitions of equal size; therefore recurrence relation for this case is;

$$T(n) = 2T(n/2) + O(n)$$

By solving this recurrence, we get,

$$T(n)=O(n \log n)$$

Worst case

Quick sort gives worst case when elements are already sorted. In this case one partition contains the n-1 elements and another partition contains no element. Therefore, its recurrence relation is;

$$T(n) = T(n-1) + O(n)$$

By solving this recurrence relation, we get

$$T(n)=O(n^2)$$

Average case

All permutations of the input numbers are equally likely. On a random input array, we will have a mix of well balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree. Suppose we are alternate: Balanced, Unbalanced, Balanced.

$B(n) = 2UB(n/2) + \Theta(n)$ Balanced

$UB(n) = B(n-1) + \Theta(n)$ Unbalanced

Solving: $B(n) = 2(B(n/2-1) + \Theta(n/2)) + \Theta(n)$

$= 2B(n/2-1) + \Theta(n)$

$= \Theta(n \log n)$

Merge Sort

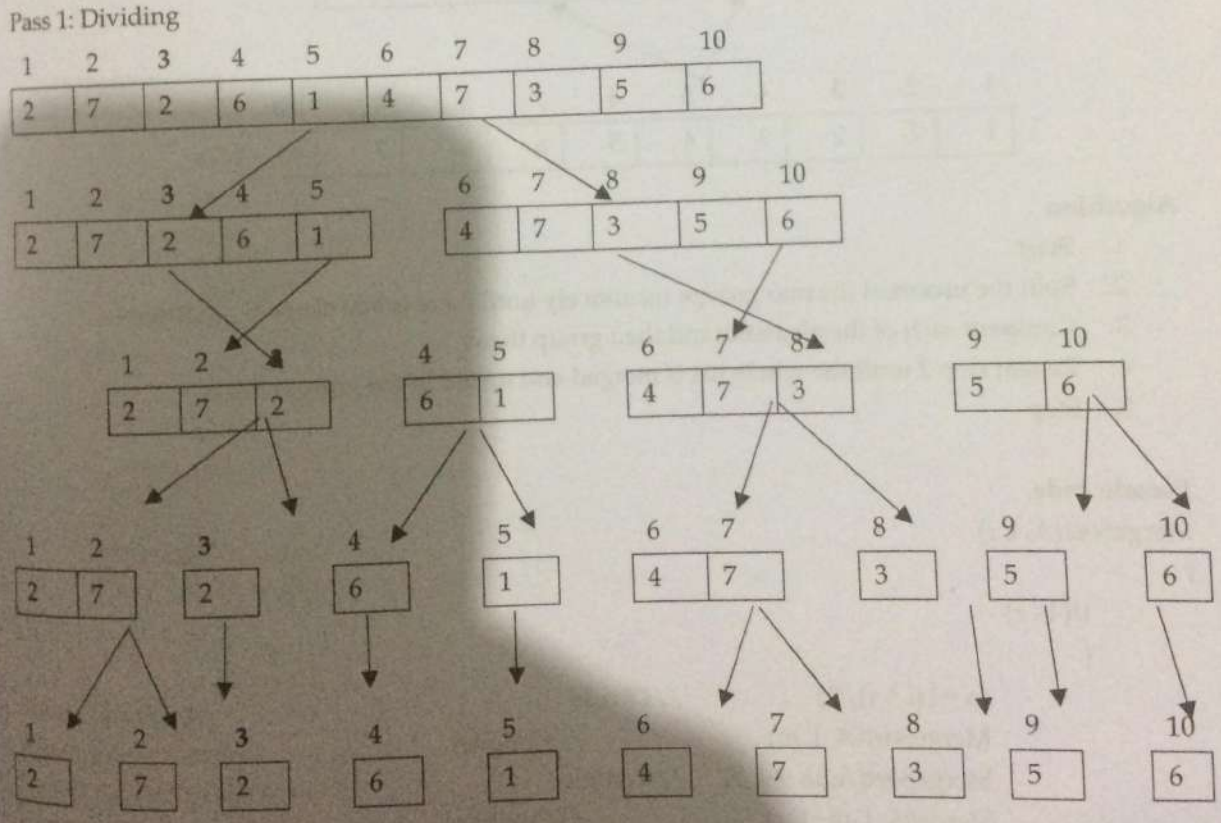
Merge sort is an efficient sorting algorithm which involves merging two or more sorted files into a third sorted file. Merging is the process of combining two or more sorted files into a third sorted file. The merge sort algorithm is based on divide and conquer method.

The process of merge sort can be formalized into three basic operations.

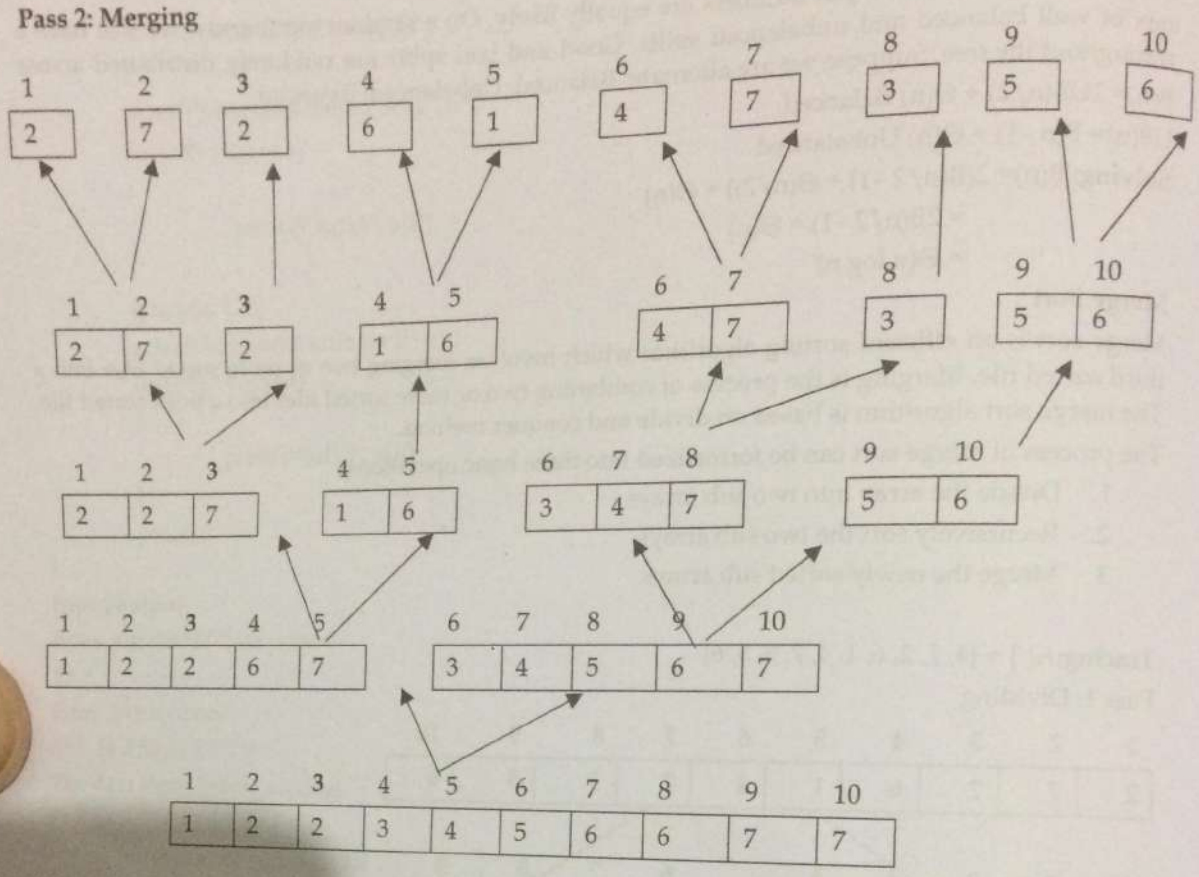
1. Divide the array into two sub arrays
2. Recursively sort the two sub arrays
3. Merge the newly sorted sub arrays

Tracing: A[ ] = {4, 7, 2, 6, 1, 4, 7, 3, 5, 6}

Pass 1: Dividing



Pass 2: Merging



Algorithm

1. Start
2. Split the unsorted list into groups recursively until there is one element per group
3. Compare each of the elements and then group them
4. Repeat step 2 until the whole list is merged and sorted in the process
5. Stop

Pseudo code

```
MergeSort(A, l, r)
{
    If( l< r)
    {
        m = ⌊(l + r)/2⌋              //Divide
        MergeSort(A, l, m)                  //Conquer
        MergeSort(A, m + 1, r)       //Conquer
        Merge(A, l, m+1, r)                 //Combine
    }
}
```

```
Merge(A, B, l, m, r)
{
        x=l;
        y=m;
        k=l;
        while(x<m && y<r)
        {
                if(A[x] < A[y])
                {
                        B[k]= A[x];
                        k++;
                        x++;
                }
                else
                {
                        B[k] = A[y];
                        k++;
                        y++;
                }
        }
        while(x<m)
        {
                A[k] = A[x];
                k++;
                x++;
        }
        while(y<r)
        {
                A[k] = A[y];
                k++;
                y++;
        }
        for(i=l; i<= r; i++)
                A[i] = B[i]
}
```

Time Complexity

No of sub-problems=2

Size of each subproblem=n/2

Dividing cost=constant

Merging cost=n

Thus recurrence relation for Merge sort is;

$T(n) = 1$ if $n=1$

$T(n) = 2\,T(n/2) + O(n)$ if $n>1$

By solving this recurrence relation, we get,

Time Complexity = $T(n) = O(n \log n)$

Complete program in C for Merge sort

```
#include<stdio.h>
#include<conio.h>
void merge(int a[ ], int l, int m, int r)
{
        int x=l;
        int k=l, i;
        int b[10];
        int y=m;
        while(x<m && y<=r)
        {
                if(a[x]<a[y])
                {
                    b[k]=a[x];
                        k++;
                        x++;
                }
                else{
                        b[k]=a[y];
                        k++;
                        y++;
                }
        }
        while(x<m)
        {
                b[k]=a[x];
                k++;
                x++;
        }
        while(y<=r)
        {
                b[k]=a[y];
                y++;
                k++;
        }
        for(i=l; i<=r; i++)
        {
```

```
            a[i]=b[i];
        }
}

void merge_sort(int a[ ], int l, int r)
{
        int mid;
        if(l<r)
        {
                mid=(l+r)/2;
                merge_sort(a, l, mid);
                merge_sort(a, mid+1, r);
                merge(a, l, mid+1, r);
        }
}
void main( )
{
        int a[100], n, i, l, r;
        printf("Enter no of elements\n");
        scanf("%d", &n);
        printf("Enter %d elements", n);
        l=0;
        r=n-1;
        for(i=0; i<n; i ++)
        {
                scanf("%d", &a[i]);
        }
        printf("elements before sort:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
        merge_sort(a, l, r);
        printf("\n elements after sort:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
        getch( );
}
```

Input/output

Enter number of data points

10
Enter 10 numbers
69 5 44 3 55 45 2 1 7 9
The data items before sorting:
69 5 44 3 55 45 2 1 7 9
The data items after sorting:
1 2 3 5 7 9 44 45 55 69

Shell sort

The first algorithm to improve on the insertion sort substantially was **Shell sort**, which was discovered in 1959 by Donald Shell. Though it is not the fastest algorithm known, **Shell sort** is a sub quadratic algorithm whose code is only slightly longer than the insertion sort, making it the simplest of the faster algorithms. Shells idea was to avoid the large amount of data movement, first by comparing elements that were far apart and then by comparing elements that were less far apart, and so on, gradually shrinking toward the basic insertion sort.

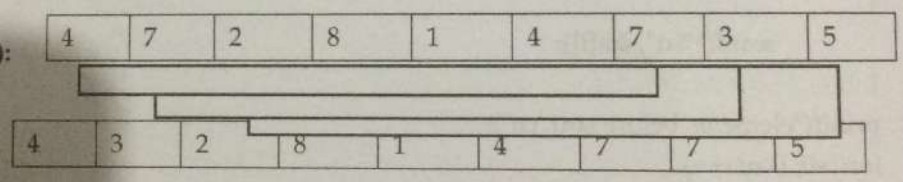
The shell sort is a diminishing increment sort. This sort divides the original file into separate sub-files. These sub-files contain every k^{th} element of the original file. The value of k is called increment.

For example, if there are n elements to be sorted and value of k is five then,

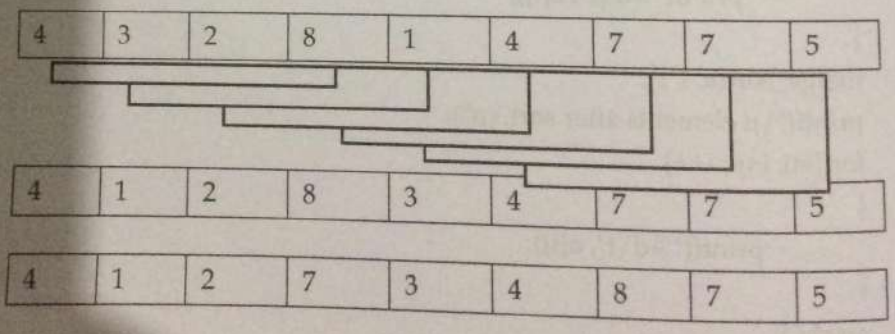
Sub-file 1→a[0], a[5], a[10], a[15].....
Sub-file 2→a[1], a[6], a[11], a[16].....
Sub-file 3→a[2], a[7], a[12], a[17].....
Sub-file 4→a[3], a[8], a[13], a[18].....
Sub-file 5→a[4], a[9], a[14], a[19].....

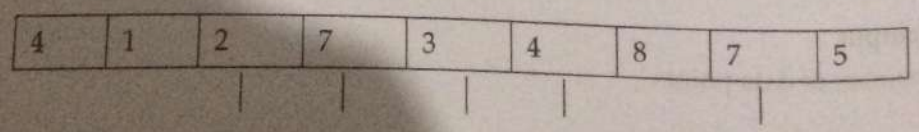
Tracing: A[ ] = {4, 7, 2, 8, 1, 4, 7, 3, 5}

Pass 1:
Increment=6(say):



Pass 2:
Increment=6/2=3:





Pass 3:
Increment=3/2=1:

1	2	3	4	4	5	7	7	8

Complete program in C for Shell sort

```
void main()
{
        int array[100],n;
        int  k, i, j, increment, temp;
        printf("Enter number of elements\n");
        scanf("%d", &n);
        for (increment = n/2; increment > 0; increment /=2)
        {
        for (i = increment; i < n; i++)
        {
                        temp = array[i];
                for (j = i; j >= increment; j -= increment)
                        {
                        if (temp < array[j - increment])
                                {
                                        array[j] = array[j - increment];
                        }
                                else
                                {
                                        break;
                        }
                }
                        array[j] = temp;
        }
        }
        printf("After Sorting:");
        for (k = 0; k < 5; k++)
        {
                printf("%d\t", array[k]);
        }
    }
}
```

Input/output

Enter number of data points
10

Enter 10 numbers
69 5 44 3 55 45 2 1 7 9
The data items before sorting:
69 5 44 3 55 45 2 1 7 9
The data items after sorting:
1 2 3 5 7 9 44 45 55 69

Radix sort

Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes. Intuitively, one might want to sort numbers on their most significant digit. However, Radix sort works counter-intuitively by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

Key points of radix sort algorithm:

1. Radix Sort is a linear sorting algorithm.
2. Time complexity of Radix Sort is O(n d), where **n** is the size of array and **d** is the number of digits in the largest number.
3. It is not an **in-place sorting** algorithm as it requires extra additional space.
4. Radix Sort is **stable sort** as relative order of elements with equal values is maintained.
5. Radix sort **can be slower** than other sorting algorithms like merge sort and quick sort, if the operations are not efficient enough. These operations include inset and delete functions of the sub-list and the process of isolating the digits we want.
6. Radix sort is less flexible than other sorts as it depends on the digits or letter.

Algorithm

```
Radix-Sort (list, n)
{
        shift = 1
        for loop = 1 to keysize do
                for entry = 1 to n do
                        bucketnumber = (list[entry].key / shift) mod 10
                        append (bucket[bucketnumber], list[entry])
                list = combinebuckets()
                shift = shift * 10
}
```

Analysis

Each key is looked at once for each digit or letter if the keys are alphabetic of the longest key. Hence, if the longest key has **m** *digits and there are* **n** *keys, radix sort has order* **O(m.n)**. *However, if we look at these two values, the size of the keys will be relatively small when compared to the number of*

keys. For example, if we have six-digit keys, we could have a million different records. Here, we see that the size of the keys is not significant, and this algorithm is of linear complexity **O(n)**.

Example: Assume the input array is:

{10, 21, 17, 34, 44, 11, 654, 123}

Based on the algorithm, we will sort the input array according to the **one's digit** (least significant digit).

0	10
1	21, 11
2	
3	123
4	34, 44, 654
5	
6	
7	17
8	
9	

So, the array becomes 10, 21, 11, 123, 34, 44, 654, 17

Now, we'll sort according to the **ten's digit**:

0	
1	10, 11, 17
2	21, 123
3	34
4	44
5	654
6	
7	
8	
9	

Now, the array becomes: 10, 11, 17, 21, 123, 34, 44, 654

Finally, we sort according to the **hundred's digit** (most significant digit):

0	010, 011, 017, 021, 034, 044
1	123
2	
3	
4	
5	
6	654

7	
8	
9	

The array becomes: [10, 11, 17, 21, 34, 44, 123, 654] which is sorted.

C function for Radix sort

```
void countsort(int arr[ ], int n, int place)
{
    int i, freq[range]={0};     //range for integers is 10 as digits range from 0-9
    int output[n];
    for(i=0; i<n; i++)
        freq[(arr[i]/place)%range]++;
    for(i=1; i<range; i++)
        freq[i]+=freq[i-1];
    for(i=n-1; i>=0; i--)
    {
        output[freq[(arr[i]/place)%range]-1]=arr[i];
        freq[(arr[i]/place)%range]--;
    }
    for(i=0; i<n; i++)
        arr[i]=output[i];
}
void radixsort(int arr[ ], int n, int maxx)  //maxx is the maximum element in the array
{
    int mul=1;
    while(maxx)
    {
        countsort(arr, n, mul);
        mul*=10;
        maxx/=10;
    }
}
```

Comparison of various sorting algorithms

In brief the worst, best and average case complexities of various sorting algorithms are tabulated below;

Sorting technique	Worst case	Average case	Best case	Comment
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$	
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Unstable
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Require extra memory

Heap sort	O(n log n)	O(n log n)	O(n log n)	Large constant
Quick sort	$O(n^2)$	O(n log n)	O(n log n)	Small constant

Searching

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

Sequential Search

Linear search algorithm finds given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means,.the search element is compared with element by element in the list.

Algorithm

1. Start
2. Read the search element from the user
3. Compare, the search element with the first element in the list.
4. If both are matching, then display "Given element found!!!" and terminate the function
5. If both are not matching, then compare search element with the next element in the list.
6. Repeat steps 4 and 5 until the search element is compared with the last element in the list.
7. If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function
8. Stop

Pseudo code

```
LinearSearch(A, n, key)
{
        flag=0;
        for(i=0; i<n; i++)
        {
                if(A[i] == key)
                        flag=1;
        }
        if(flag==1)
                Print "Search successful"
        else
                Print "Search un-successful"
```

```
}
```

Analysis

Time complexity = O(n)

Complete C program for Sequential search

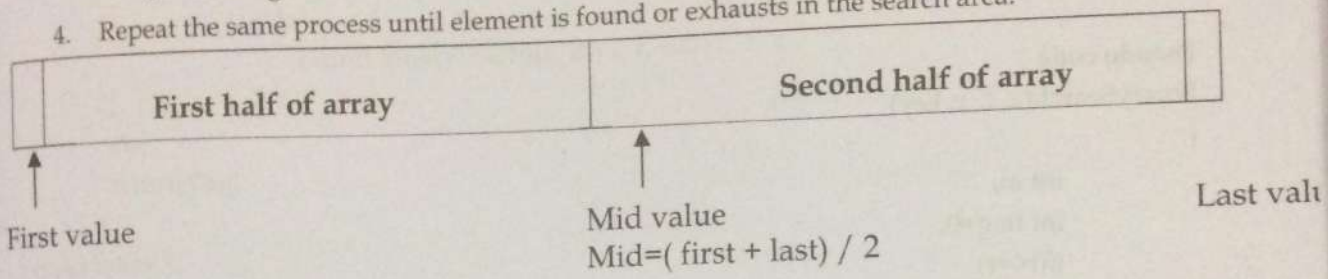
```
#include <stdio.h>
int main()
{
        int a[100], key, i, n;
        printf("Enter the number of elements in array\n");
        scanf("%d", &n);
        printf("Enter %d numbers \n", n);
        for (i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        printf("Enter the number to search\n");
        scanf("%d", &key);
        for (i = 0; i < n; i++)
        {
                if (a[i] == key)
                {
                        printf("%d is present at location %d.\n", key, i+1);
                        break;
                }
        }
        if (i == n)
                printf("%d is not present in array.\n", key);
        return 0;
}
```

Binary Search

Binary search algorithm finds given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in an order. The binary search cannot be used for list of element which is in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sub list of the middle element. If the search element is larger, then we repeat the same process for right sub list of the middle element. We repeat this process until we find the search element in the list or until we left with a sub list of only one element.

And if that element also doesn't match with the search element, then the result is "Element not found in the list". The logic behind this technique is given below:

1. First find the middle element of the array
2. Compare the middle element with an item.
3. There are three cases:
   a. If it is a desired element then search is successful
   b. If it is less than desired item then search only the first half of the array.
   c. If it is greater than the desired element, search in the second half of the array.
4. Repeat the same process until element is found or exhausts in the search area.



Tracing

Search for key= 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}

Initially

-1	5	6	18	19	25	46	78	102	114
0	1	2	3	4	5	6	7	8	9

Step 1: Since [middle element (19) > key (6)]

-1	5	6	18	19	25	46	78	102	114
0	1	2	3	4	5	6	7	8	9

Step 2: Since [middle element (5) < key (6)]

~~-1~~	~~5~~	6	18	~~19~~	~~25~~	~~46~~	~~78~~	~~102~~	~~114~~
~~0~~	~~1~~	2	3	4	~~5~~	~~6~~	~~7~~	~~8~~	~~9~~

Step 1: Since [middle element (6) > key (6)]

~~-1~~	~~5~~	6	~~18~~	~~19~~	~~25~~	~~46~~	~~78~~	~~102~~	~~114~~
~~0~~	~~1~~	2	~~3~~	4	~~5~~	~~6~~	~~7~~	~~8~~	~~9~~

Algorithm

1. Start
2. Read the search element from the user
3. Find the middle element in the sorted list
4. Compare, the search element with the middle element in the sorted list.
5. If both are matching, then display "Given element found!!!" and terminate the function
6. If both are not matching, then check whether the search element is smaller or larger than middle element.

7. If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sub list of the middle element.
8. If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sub list of the middle element.
9. Repeat the same process until we find the search element in the list or until sub list contains only one element.
10. If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.
11. Stop

Pseudo code

```
BinarySearch(a, l,  r, key)
{
        int m;
        int flag=0;
        if(l<=r)
        {
                m =(l + r)/2 ;
                if(key==a[m])
                    flag=m;
                else if (key<a[m])
                    return BinarySearch(a, l, m-1, key) ;
                else
                    return BinarySearch(a, m+1, r, key) ;
        }
        else
        return flag;
}
```

Efficiency

From the above algorithm we can say that the running time of the algorithm is

$$T(n) = T(n/2) + O(1)$$

By solving this we get O (log n)

In the best case output is obtained at one run i.e. O(1) time if the key is at middle.

In the worst case the output is at the end of the array so running time is O(log n) time.

In the average case also running time is O(log n).

Complete C program for Binary search

```
#include<stdio.h>
#include<conio.h>
int BinarySearch(int a[100], int l, int r, int key)
{
        int m;
```

```
        int flag=0;
        if(l<=r)
        {
                m =(l + r)/2 ;
                if(key==a[m])
                    flag = m;
                else if (key<a[m])
                    return BinarySearch(a, l, m-1, key) ;
                else
                    return BinarySearch(a, m+1, r, key) ;
        }
        else
        return flag;
}
void main( )
{
        int n, a[100], i, key, flag;
        printf("Enter no of data items:\n");
        scanf("%d", &n);
        printf("Enter %d data items in sorted form:\n", n);
        for(i=0; i<n; i++)
                scanf("%d", &a[i]);
        printf("Enter searched item");
        scanf("%d", &key);
        flag=BinarySearch(a, 0, n-1, key);
        if(flag==0)
                printf("Search Un-Successful");
        else
                printf("Search successful and found at location %d",flag+1);
}
```

Exercise

1. Why sorting is important in computer science? Describe any one of the best sorting technique with suitable example.
2. What is stable sort? List out any two stable sorting techniques with example.
3. What is sorting? How it is differ from searching?
4. Which sorting technique gives worst case when elements are in already sorted form?
5. Here is an array of ten integers:

   3 8 9 1 7 0 2 6 4

   Draw this array after the first iteration of the large loop in a selection sort (sorting from smallest to largest).
6. Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

Draw this array after the first iteration of the large loop in an insertion sort (sorting from smallest to largest). This iteration has shifted at least one item in the array!

7. Suppose that you are writing a program that has this selection sort static method available:

   a. **void selection_sort(int[ ] data, int first, int n);**

   Your program also has an integer array called x, with 10 elements. Write two method activations: The first activation uses selection sort to sort all of x; the second call uses selection sort to sort x[3].....x[9].

8. Here is an array of ten integers:

   a. 5 3 8 9 1 7 0 2 6 4

   Suppose we partition this array using quick sort's partition function and using 5 for the pivot. Draw the resulting array after the partition finishes.

9. Here is an array of ten integers:

   a. 5 3 8 9 1 7 0 2 6 4

   Draw this array after the two recursive calls of merge sort are completed, and before the final merge step has occurred.

10. Some sorting methods, like heap sort and array-based quick sort, are not naturally stable. Suggest a way to make _any_ sorting algorithm stable by extending the keys (making them longer and adding extra information).

11. Suppose we have a O(n) time algorithm that finds median of an unsorted array. Now consider a Quick Sort implementation where we first find median using the above algorithm, then use median as pivot. What will be the worst case time complexity of this modified Quick Sort?

12. Given an unsorted array. The array has this property that every element in array is at most k distance from its position in sorted array where k is a positive integer smaller than size of array. Which sorting algorithm can be easily modified for sorting this array and what is the obtainable time complexity?

13. Here is an array of ten integers:

    [5 3 8 9 1 7 0 2 6 4]

    Draw this array after the FIRST iteration of the large loop in an insertion sort (sorting from smallest to largest). This iteration has shifted at least one item in the array!

14. Describe a case where quick sort will result in quadratic behavior.

15. Here is an array which has just been partitioned by the first step of quick sort:

    [3, 0, 2, 4, 5, 8, 7, 6, 9]

    Which of these elements could be the pivot?

16. Write two or three clear sentences to describe how a heap sort works.

17. How does a selection sort work for an array? Explain with suitable example.

18. What is the maximum number of comparisons needed to sort 7 items using radix sort? (assume each item is 4 digit decimal number)

19. Which of the sorting methods will be the best if number of swapping done, is the only measure of efficiently?

20. Sort following data items by using a). Heap sort b). Quick sort c). Merge sort.

    [4, 5, 6, 88, 1, 22, 43, 56, 53, 2, 11, 57, 9]

•••

Chapter

10

Graphs

Introduction

Graph is a non linear data structure; it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is an ordered pair G = (V, E) comprising a set V of vertices or nodes together with a set E of edges or links. Each edge has either one or two vertices associated with it, called its end points. Each edge is a pair (v, w), where v, w $\in$ V. Vertices are sometimes called nodes, and edges are sometimes called arcs. If the edge pair is ordered, the graph is called a **directed graph**. Directed graphs are sometimes called digraphs. In a digraph, vertex w is adjacent to vertex v if and only if (v, w) $\in$ E. Sometimes an edge has a third component, called the edge cost (or weight) that measures the cost of traversing the edge.

Example: A simple undirected graph with 7 vertices and 11 edges

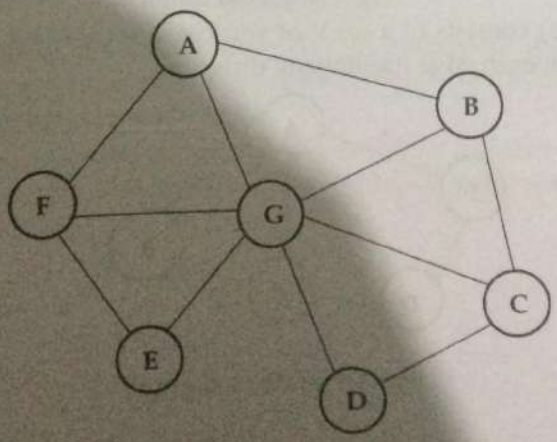


Fig: Undirected graph

Applications of graph

There are large numbers of applications of graph. We may have seen maps of a country in which airlines routes are shown as lines (arcs) connecting pairs of cities. Each line represents the edge and each city represents the node for the graph. This is a simple famous example of graph in practice.

- Graphs are used to model the geographic maps of cities in which each place in city can be represented by the node and the road connecting such places are represented by an arc (edge).
- Graphs are used to model the computer network in which each node is a machine (computer, hub, router, switch etc.) and the link between them represents the edge.
- They are used to analyze the electrical circuits, project planning, genetics etc.
- So any structured problem can be modeled by graphs. Then can help to solve typical problems those concerned with finding shortest path or most economical route between two vertices, or the smallest set of edges which connect all the vertices in a graph.

Definition of directed and undirected graphs

A directed graph (V, E) consists of a set V of vertices, a set E of **directed** edges that are ordered pairs of elements of V. In this graph loop is allowed but no two vertices can have multiple edges in same direction.

Simply a simple graph with flow of direction is called directed graph. The below figure is a directed graph.

Example:

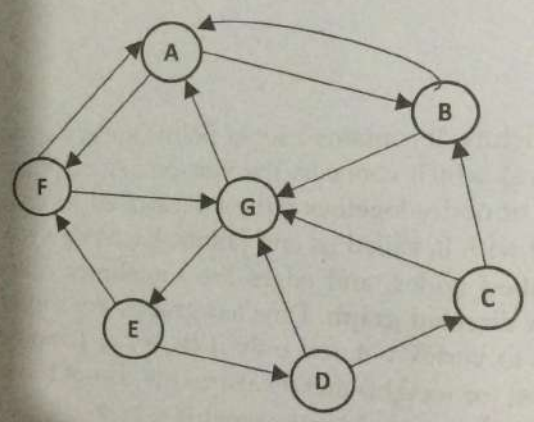


Fig: A directed graph

An undirected graph (V, E) consists of a set V of vertices, a set E of edges that are ordered pairs of elements of V. In this graph each edge has not any direction.

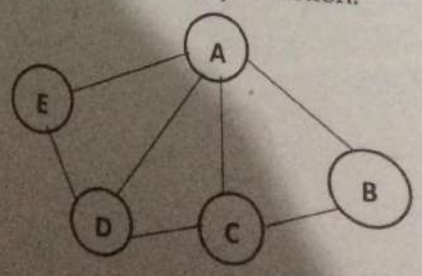


Fig: An undirected graph

Directed multi-graph

A directed multi-graph G = (V, E) consists of a set of vertices V, a set of edges E, and a function f from E to {(u, v) | u, v ∈ V}. The edges e_1 and e_2 are called multiple edges if $f(e_1) = f(e_2)$. Simply a directed graph in which two vertices may also have multiple edges in the same direction is called directed multi-graph. The figure below is an example of a directed multi-graph.

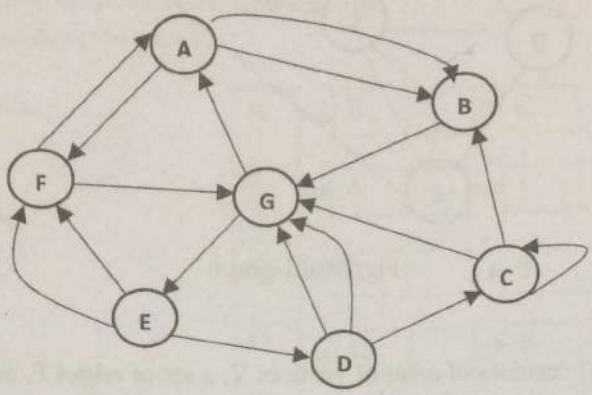


Fig: Directed multi-graph

Simple and multi-graphs

An undirected graph with no multiple edges between any two vertices or loops is called simple graph. The figure below is an example of simple graph.

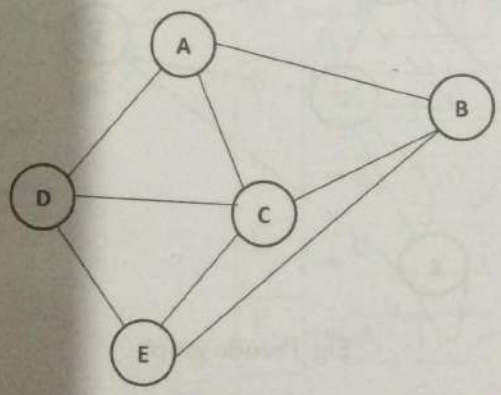


Fig: - Simple graph

A multi-graph G = (V, E) consists of a set of vertices V, a set of edges E, and a function f from E to {{u, v} | u, v ∈ V, u ≠ v}. The edges e1 and e2 are called multiple or parallel edges if f(e1) = f(e2). In this representation of graph also loops are not allowed.

Simply an undirected graph that may contain multiple edges between any two vertices but not contains loop is called multi-graph. The figure below is an example of a multi-graph.

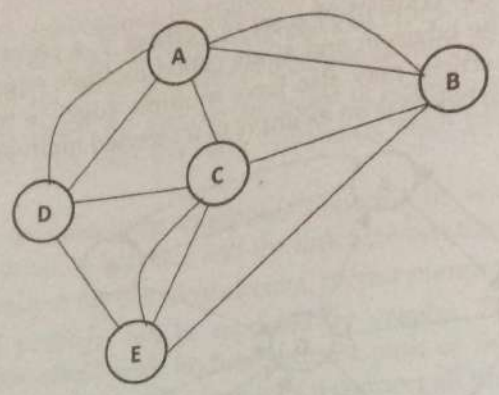


Fig: Multi-graph

Pseudo-graphs

A pseudo-graph G = (V, E) consists of a set of vertices V, a set of edges E, and a function f from E to {{u, v} | u, v ∈ V}. An edge is a loop if f(e) = {u, u} = {u} for some u ∈ V.

Simply an undirected graph with loops and multiple edges between any two vertices is called pseudo graph. The figure below is an example of a multi-graph.

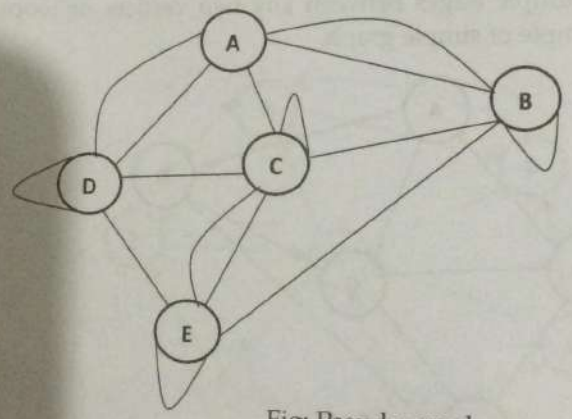


Fig: Pseudo graph

Mixed graph

A graph with both directed and undirected edges is called mixed graph.

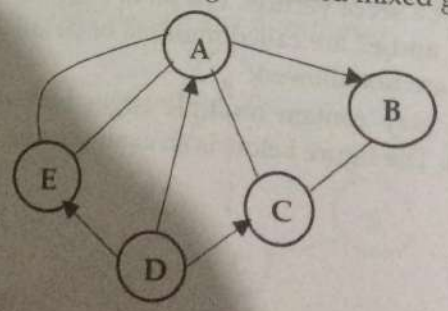


Fig: Mixed graph

Graph Representation

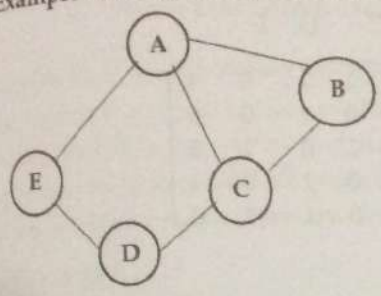
Graph is a mathematical structure and finds its applications in many areas of interest in which problems need to be solved using computer. Graph can be represented in many ways; one of the

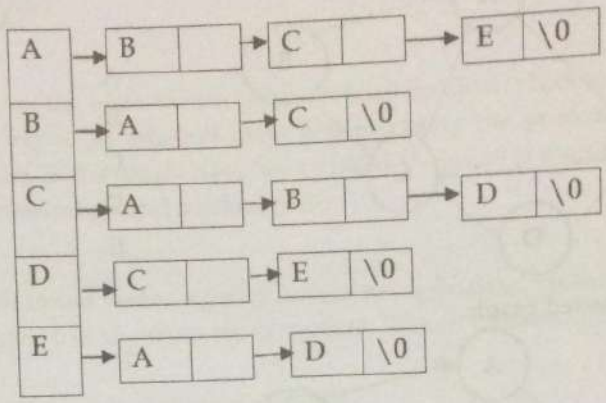
ways of representing a graph without multiple edges is by listing its edges. Some other ways are described below:

Adjacency List

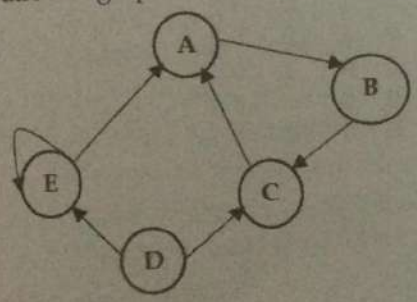
This representation of graph specifies the vertices that are adjacent to each vertex of the graph. This type of representation is suitable for the undirected graphs without multiple edges, and directed graphs. This representation looks as in the tables below.

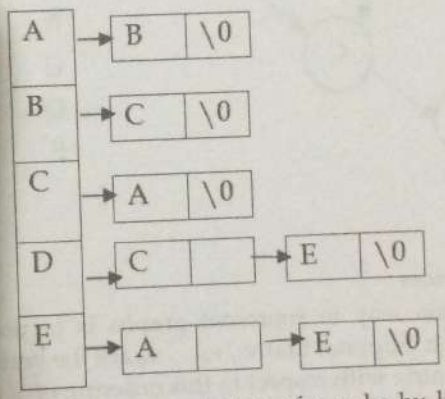
Example: Let's take an undirected graph





For directed graph





If we try to apply the algorithms of graph using the representation of graphs by lists of edges, or adjacency lists it can be tedious and time taking if there are high numbers of edges. For the sake of the computation, the graphs with many edges can be represented in other ways. In this class we discuss two ways of representing graphs in form of matrix.

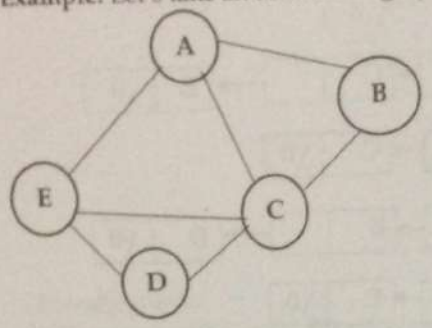
Adjacency matrix representation of graph

The adjacency lists, can be cumbersome if there are many edges in the graph. To simplify computation, graphs can be represented using matrices. Two types of matrices commonly used to represent graphs will be presented here. One is based on the adjacency of vertices, and the other is based on incidence of vertices and edges. Suppose that G = (V, E) is a simple graph where |V| = n. Suppose that the vertices of G are listed arbitrarily as $v_1, v_2, \ldots v_n$. The adjacency matrix A of G, with respect to this listing of the vertices, is the n x n zero-one matrix with 1 as its $(i, j)^{th}$ entry when vi and vj are adjacent, and 0 as its $(i, j)^{th}$ entry when they are not adjacent.

In other words, if its adjacency matrix is $A = [a_{ij}]$, then

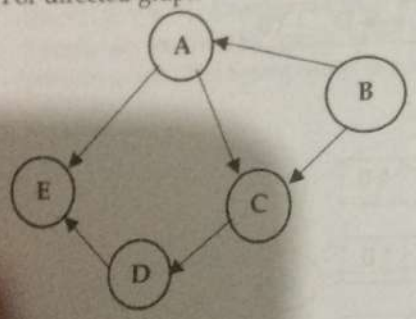
$$a_{ij} = \begin{cases} 1 & \text{if } \{vi, vj\} \text{ is an edge of G,} \\ 0 & \text{otherwise} \end{cases}$$

Example: Let's take an undirected graph



$$\begin{array}{c|c} & \begin{matrix} A & B & C & D & E \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

For directed graph



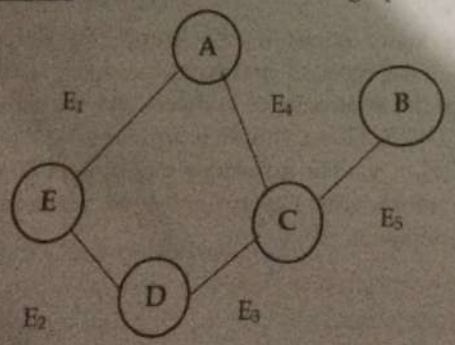
$$\begin{array}{c|c} & \begin{matrix} A & B & C & D & E \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} & \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Incidence Matrices

Another common way to represent graphs is to use incidence matrices. Let G = (V, E) be an undirected graph. Suppose that $v_1, v_2, \ldots v_n$ are the vertices and $e_1, e_2, \ldots, e_m$ are the edges of G. Then the incidence matrix with respect to this ordering of V and E is the $n \times m$ matrix $M = [m_{ij}]$, where m_{ij} =1 when edge ej is incident with v_i, 0 otherwise.

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise} \end{cases}$$

Example: Let's take an undirected graph



$$\begin{array}{c|c} & \begin{matrix} E1 & E2 & E3 & E4 & E5 \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} & \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Graph Traversals

Traversing a graph means visiting all the vertices in a graph exactly one. In tree traversal, there is a special node root, from which the traversal starts. But in graph all nodes are treated equally. So the starting node in graph can be chosen arbitrarily. In general, for diagrammatic illustration, the left-up node is taken as starting vertex for traversal. Two common approaches for traversal are:

- Breadth First Search Traversal (BFS)
- Depth First Search Traversal (DFS)

Breadth First Search Traversal (BFS)

The traversal starts at a node v, after marking the node the traversal visits all incident edges to node v after marking the nodes and then moving to an adjacent node and repeating the process. The traversal continues until all unmarked nodes in the graph have been visited. A queue is maintained in the technique to maintain the list of incident edges and marked nodes.

Steps in BFS

Let, G = (V, E) is a graph or digraph with initial vertex v_0; each vertex has a Boolean visited field initialized to false; T is an empty set of edges; L is an empty list of vertices. Then it follows following steps:

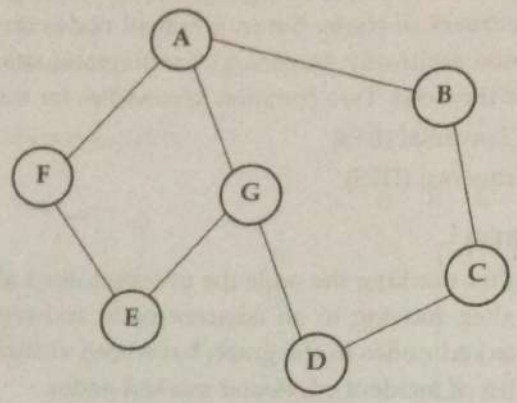
1. Initialize an empty queue Q for temporary storage of vertices.
2. Enqueue v_0 into Q.
3. Repeat steps 4–6 while Q is not empty.
4. Dequeue Q and set such item into v.
5. Add x to vertex list, L.
6. Do step 7 for each vertex w that is adjacent to v.
7. If y has not been visited, do steps 8–9.
8. Add the edge vw to edge list, T.
9. Enqueue y into queue, Q.

Algorithm

```
BFS (G, s)          //s is start vertex
{
        T = {s};
        L =Φ;               //an empty queue
        Enqueue (L, s);
        while (L != Φ )
        {
                v = Dequeue (L);
                For each neighbor w to v
                if ( w∉ L && w∉ T )
                {
                        Enqueue (L, w);
                        T = T U {w};            //put edge {v, w} also
                }
        }
}
```

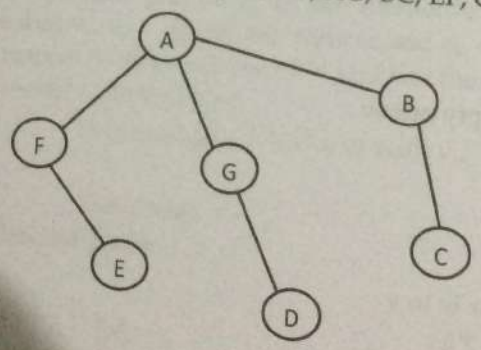
Tracing of BFS algorithm

let's take start vertex is v_0 = A

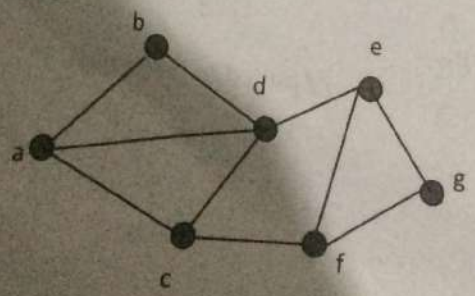


Queue (Q)	Dequeue item (v)	Set of vertices (L)	Adjacent vertex(w)	Edge list (T)
A	.....	.....	.....	.....
.....	A	A	B	AB
B	.....	A	F	AB, AF
B F	.....	A	G	AB,AF,AG
BFG	B	A, B	C	AB,AF,AG,BC
FGC	F	A,B,F	E	AB,AF,AG,BC,EF
GCE	G	A,B,F,G	D	AB,AF,AG,BC,EF,GD
CED	C	A,B,F,G,C	.....	AB,AF,AG,BC,EF,GD
ED	E	A,B,F,G,C,E	.....	AB,AF,AG,BC,EF,GD
D	D	A,B,F,G,C,E,D	.....	AB,AF,AG,BC,EF,GD
.....	.....	A,B,F,G,C,E,D	.....	AB,AF,AG,BC,EF,GD

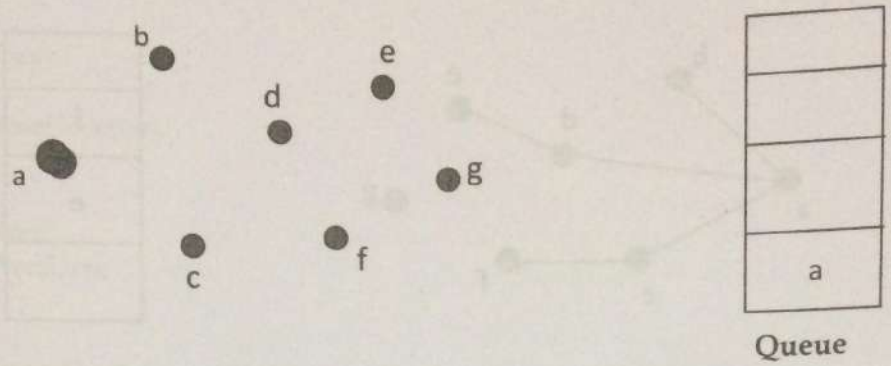
The resulting BFS order of visitation is returned in the list L = (A, B, F, G, C, E, D), and the resulting BFS spanning tree is returned in the set T = {AB, AF, AG, BC, EF, GD}.



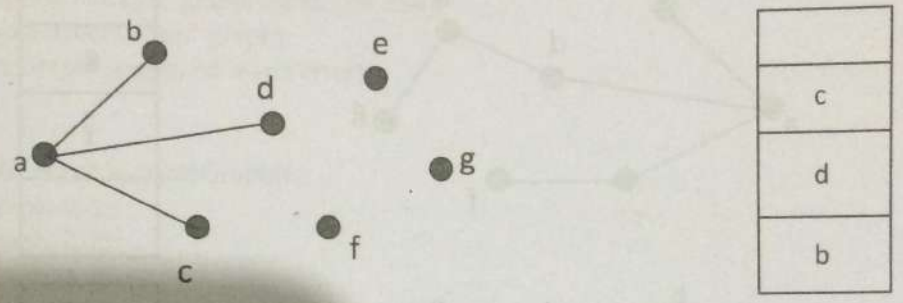
Tracing method 2 for BFS



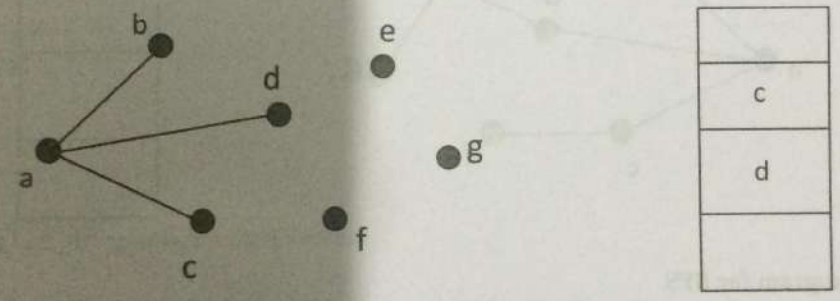
Step 1:



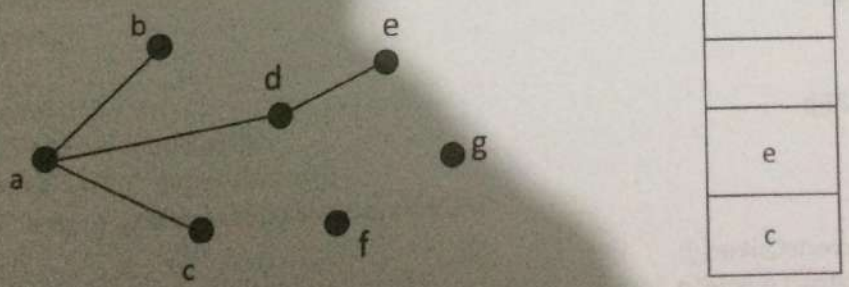
Step 2:



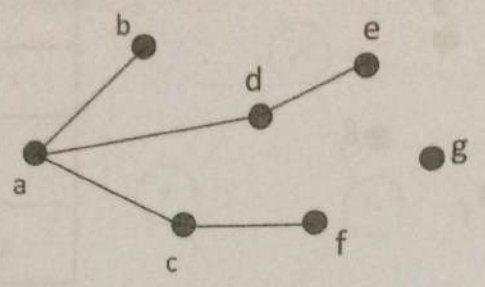
Step 3:

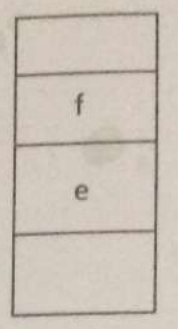


Step 4:

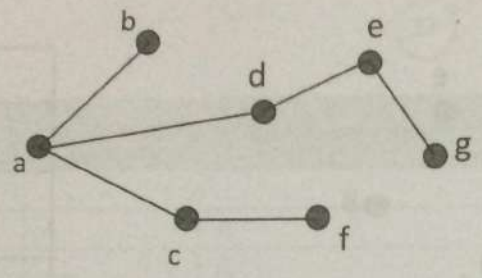


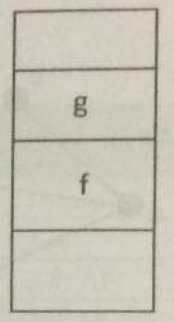
Step 4:



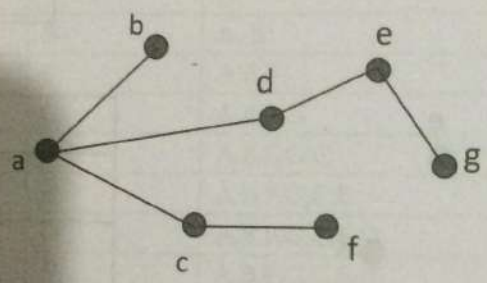


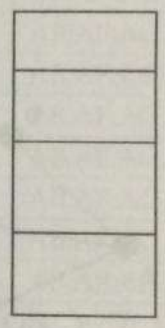
Step 5:





Step 6:





Complete C program for BFS

```
include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue
{
  int items[SIZE];
  int front;
  int rear;
};
struct queue* createQueue( );
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);
struct node
```

```
{
  int vertex;
  struct node* next;
};
struct node* createNode(int);
struct Graph
{
  int numVertices;
  struct node** adjLists;
  int* visited;
};
struct Graph* createGraph(int vertices);
void addEdge(struct Graph* graph, int src, int dest);
void printGraph(struct Graph* graph);
void bfs(struct Graph* graph, int startVertex);
int main()
{
  struct Graph* graph = createGraph(6);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 1, 4);
  addEdge(graph, 1, 3);
  addEdge(graph, 2, 4);
  addEdge(graph, 3, 4);

  bfs(graph, 0);
  return 0;
}
void bfs(struct Graph* graph, int startVertex)
{
  struct queue* q = createQueue();
  graph→visited[startVertex] = 1;
  enqueue(q, startVertex);
  while(!isEmpty(q))
  {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);
    struct node* temp = graph→adjLists[currentVertex];
    while(temp)
    {
      int adjVertex = temp→vertex;
      if(graph→visited[adjVertex] == 0)
      {
        graph→visited[adjVertex] = 1;
        enqueue(q, adjVertex);
```

```
        }
        temp = temp→next;
      }
    }
  }

  struct node* createNode(int v)
  {
    struct node* newNode = malloc(sizeof(struct node));
    newNode→vertex = v;
    newNode→next = NULL;
    return newNode;
  }

  struct Graph* createGraph(int vertices)
  {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph→numVertices = vertices;
    graph→adjLists = malloc(vertices * sizeof(struct node*));
    graph→visited = malloc(vertices * sizeof(int));


    int i;
    for (i = 0; i < vertices; i++)
    {
      graph→adjLists[i] = NULL;
      graph→visited[i] = 0;
    }
    return graph;
  }

  void addEdge(struct Graph* graph, int src, int dest)
  {
    struct node* newNode = createNode(dest); // Add edge from src to dest
    newNode→next = graph→adjLists[src];
    graph→adjLists[src] = newNode;
    newNode = createNode(src);    // Add edge from dest to src
    newNode→next = graph→adjLists[dest];
    graph→adjLists[dest] = newNode;
  }

struct queue* createQueue()
{
  struct queue* q = malloc(sizeof(struct queue));
  q→front = -1;
  q→rear = -1;
  return q;
```

```
}
int isEmpty(struct queue* q)
{
  if(q→rear == -1)
    return 1;
  else
    return 0;
}

void enqueue(struct queue* q, int value)
{
  if(q→rear == SIZE-1)
    printf("\n Queue is Full!!");
  else
  {
    if(q→front == -1)
        q→front = 0;
    q→rear++;
    q→items[q→rear] = value;
  }
}
int dequeue(struct queue* q)
{
  int item;
  if(isEmpty(q))
  {
    printf("Queue is empty");
    item = -1;
  }
  else
  {
        item = q→items[q→front];
        q→front++;
        if(q→front > q→rear)
        {
        printf("Resetting queue");
                q→front = q→rear = -1;
        }
  }
  return item;
}
void printQueue(struct queue *q)
{
  int i = q→front;
  if(isEmpty(q))
  {
```

```
        printf("Queue is empty");
    }
    else
    {
        printf("\n Queue contains \n");
        for(i = q→front; i < q→rear + 1; i++)
            printf("%d ", q→items[i]);
    }
}
```

Depth First Traversal (DFS)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

A depth first search of an arbitrary graph can be used to perform a traversal of a general graph. The technique picks up a node and marks it. An unmarked adjacent node to previous node is then selected and marked, becomes the new start node, possibly leaving the previous node with unexplored edges for the present. The traversal continued recursively, until all unmarked nodes of the current path are visited. The process is continued for all the paths of the graph. If this cannot be done, move back to another vertex and repeat the process. The whole process is continued until all the vertices are met. This method of search is also called backtracking. A Stack data structure is maintained in the technique to maintain the list of incident edges and marked nodes.

Algorithm

1. Start
2. Define a Stack of size total number of vertices in the graph.
3. Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
4. Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
5. Repeat step 4 until there are no new vertex to be visit from the vertex on top of the stack.
6. When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.
7. Repeat steps 4, 5 and 6 until stack becomes Empty
8. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph
9. Stop

Pseudo code

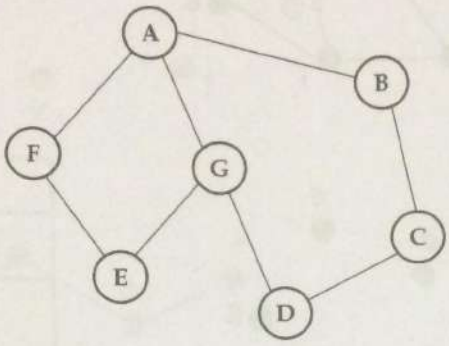
Recursive steps for DFS are shown in below:

```
DFS (G, s)
{
        T={s};
        Traverse (s);
}
Traverse (v)
{
        for each w adjacency to v and not yet in T
        {
                T=T U {w};        // put edge {v, tw} also
```

```
            Traverse (w);
        }
}
```

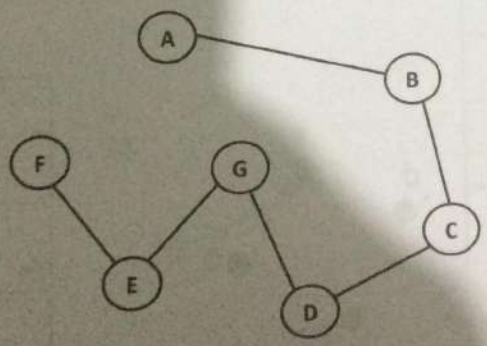
Tracing

The start vertex is v_0 = A

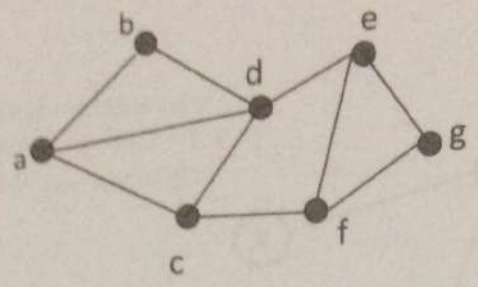


Set of vertices (L)	Stack (S)	Popped element of Stack (v)	Adjacent vertex (w)	Edge list (T)
A	A	.....	.....	.....
.....	.....	A	B	AB
A,B	B	B	C	AB, BC
A,B,C	C	C	D	AB,BC,CD
A,B,C,D	D	D	G	AB,BC,CD,DG
A,B,C,D,G	G	G	E	AB,BC,CD,DG,GE
A,B,C,D,G,E	E	E	F	AB,BC,CD,DG,GE,EF
A,B,C,D,G,E,F	F	F	.....	AB,BC,CD,DG,GE,EF
A,B,C,D,G,E,F	.....	.....	.....	AB,BC,CD,DG,GE,EF

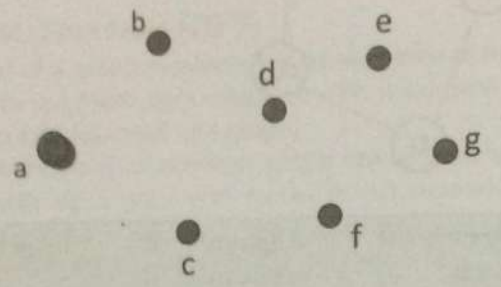
The resulting BFS order of visitation is returned in the list L = (A, B, C, D, G, E, F), and the resulting DFS spanning tree is returned in the set T = {AB, BC, CD, DG, GE, EF}

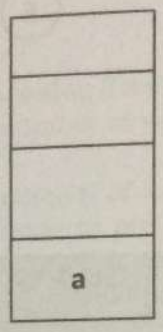


Tracing method 2 for DFS



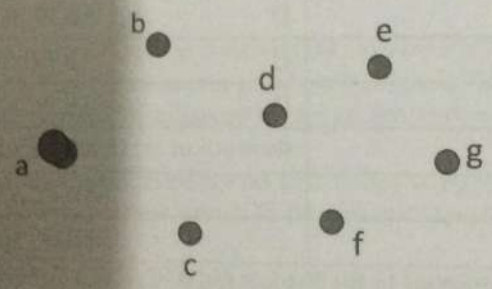
Step 1:

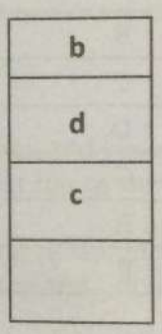




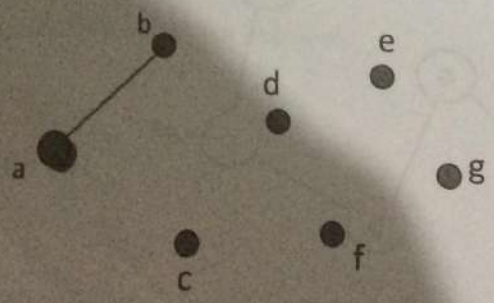
Stack

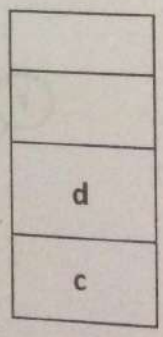
Step 2:



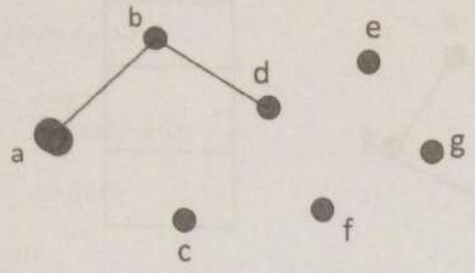


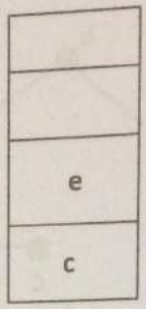
Step 3:



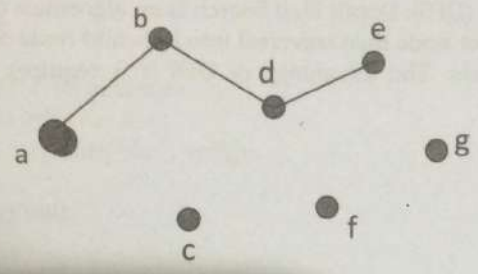


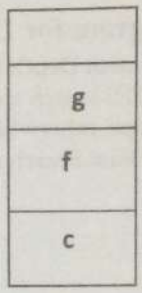
Step 4:



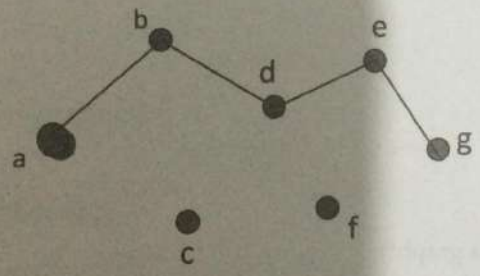


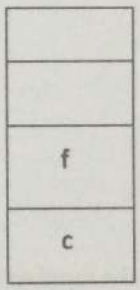
Step 5:



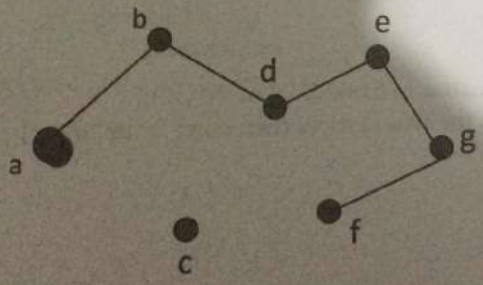


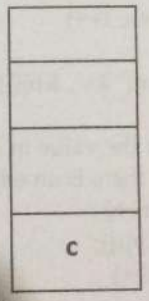
Step 6:



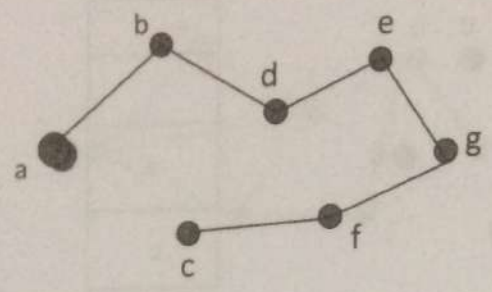


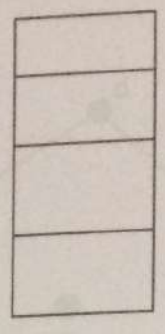
Step 7:





Step 8:





Complete C program for DFS

C program to implement Depth First Search (DFS). Depth First Search is an algorithm used to search the Tree or Graph. DFS search starts from root node then traversal into left child node and continues, if item found it stops otherwise it continues. The advantage of DFS is it requires less memory compare to Breadth First Search (BFS).

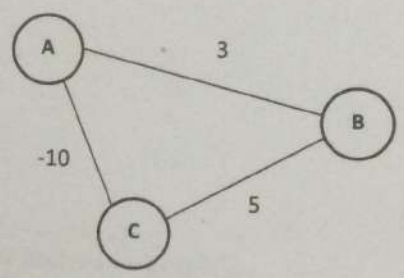
```
#include<stdio.h>
#include<conio.h>
char stack[20];
int top=-1, n;
char arr[20];
char dfs(int );
char ajMat[20][20];
char b[20];
void display();
int p=0;
int main()
{
        char v;
        int l=0, i, j;
        char k=b[0];
        printf("Enter the number of nodes in a graph");
        scanf("%d", &n);
        printf("Enter the value of node of graph");
        for(int i=0; i<n; i++)
        {
                scanf("%s", &b[i]);
        }
        printf("Enter the value in adjacency matrix in from of 'Y' or 'N'\n");
        printf("\n If there is an edge between the two vertices then enter 'Y' or 'N'\n");
        for(i=0; i<n; i++)
        printf(" %c ",b[i]);
        for(i=0;i<n; i++)
        {
                printf("\n%c ", b[i]);
                for(j=0; j<n; j++)
                {
                printf("%c ", v=getch());
                ajMat[i][j]=v;
                }
```

```
            printf("\n\n");
        }
        for(int i=0;i<n;i++)
        {
            l=0;
            while(k!=b[l])
                l++;
            k=dfs(l);
        }
        display();
        getch();
}

void display()
{
        printf(" DFS of Graph : ");
        for(int i=0; i<n; i++)
            printf("%c ", arr[i]);
}
void push(char val)
{
        top=top+1;
        stack[top]=val;
}

char pop()
{
        return stack[top];
}

bool unVisit(char val)
{
        for(i=0; i<p; i++)
            if(val==arr[i])
                return false;
        for(i=0; i<=top; i++)
            if(val==stack[top])
                return false;
        return true;
}

char dfs(int i)
{
    int k;
    char m;
    if(top==-1)
    {
        push(b[i]);
    }
    m=pop();
```

```
    top--;
    arr[p]=m;
    p++;
    for(int j=0; j<n; j++)
    {
        if(ajMat[i][j]=='y')
        {
        if(unVisit(b[j]))
        {
                push(b[j]);
        }
        }
    }
        return stack[top];
}
```

Shortest Paths

Given a weighted graph G = (V, E), then it has weight for every path p = <v_0, v_1..., vk_{-1}, v_k> as w(p) = w(v_0, v_1) + w(v_1,v_2) + ... + w(v_{k-1},v_k). A shortest path from u to v is the path from u to v with minimum weight. Shortest path from u to v is denoted by d(u, v). It is important to remember that the shortest path may exist in a graph or may not i.e. if there is negative weight cycle then there is no shortest path. For e.g. the below graph has no shortest path from A to C. You can notice the negative weight cycle for path A to B.



As a matter of fact, even the positive weight cycle doesn't constitute shortest path but there will be shortest path. Some of the variations of shortest path problem include:

Single Source shortest path problem

This type of problem asks us to find the shortest path from the given vertex (source) to all other vertices in a connected graph. Here only one source vertex and lot of destination vertices are take place.

Single Destination shortest path problem

This type of problem asks us to find the shortest path to the given vertex (destination) from all other vertices in a connected graph.

Single Pair shortest path problem

This type of problem asks us to find the shortest path from the given vertex (source) to another given vertex (destination). In this type of problem, we need to find shortest path form a fixed source vertex to one of the destination vertex.

All Pairs

This type of problem asks us to find the shortest path from the all vertices to all other vertices in a connected graph.

Relaxation

Relaxation of an edge (u, v) is a process of testing the total weight of the shortest path to v by going through u and if we get the weight less than the previous one then replacing the record of previous shortest path by new one.

Dijkstra's Algorithm

This is another approach of getting single source shortest paths. In this algorithm it is assumed that there is no negative weight edge. Dijkstra's algorithm works using greedy approach, as we will see later. Dijkstra's algorithm finds the shortest path from one vertex v_0 to each other vertex in a digraph. When it has finished, the length of the shortest distance from $v0$ to v is stored in the vertex v, and the shortest path from v_0 to v is recorded in the back pointers of v and the other vertices along that path.

Steps in Dijkstra's algorithm

Precondition: G = (V, w) is a weighted graph with initial vertex v_0 then it holds following steps:

1. Initialize the distance field to 0 for v0 and to for each of the other vertices.
2. Enqueue all the vertices into a priority queue Q with highest priority being the lowest distance field value.
3. Repeat steps 4–10 until Q is empty.
4. The distance and back reference fields of every vertex that is not in Q are correct
5. Dequeue the highest priority vertex into v.
6. Do steps 7–10 for each vertex w that are adjacent to v and in the priority queue.
7. Let S be the sum of the v's distance field plus the weight of the edge from v to w.
8. If s is less than w's distance field, do steps 9–10; otherwise go back to Step3.
9. Assign s to w's distance field.
10. Assign v to w's back reference field.

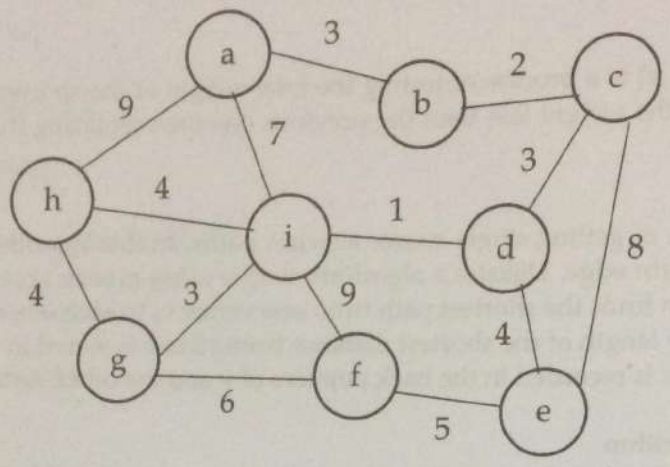
Algorithm

```
Dijkstra_Algorithm(G, w, s)
{
        for each vertex v ∈ V
                d[v] = Φ
        d[s] = 0
        S = Φ
        Q = V
        while (Q!= Φ)
        {
                u = Take minimum from Q and delete.
                S = S U{u}
                for each vertex v adjacent to u
                        if d[v] > d[u] + w(u, v)then
                        d[v] = d[u] + w(u, v)
        }
}
```

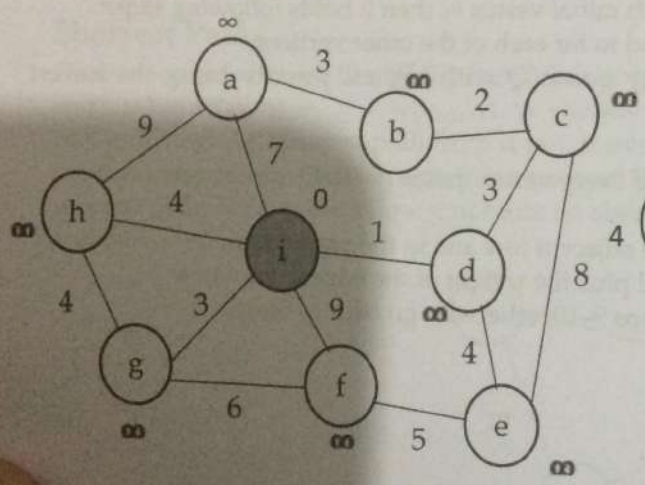
Analysis

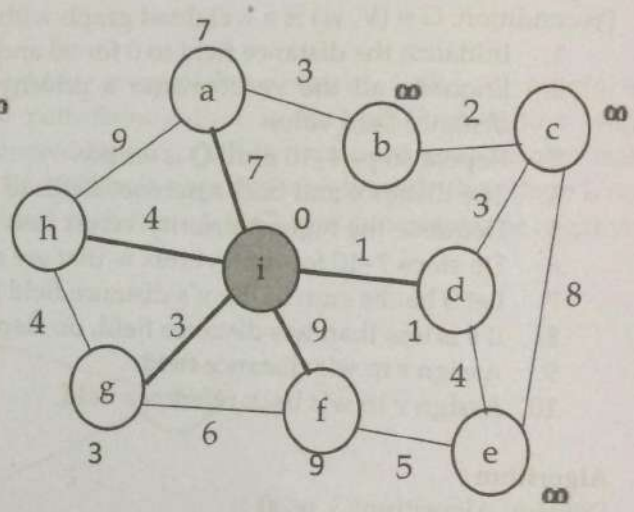
In the above algorithm, the first for loop block takes O(V) time. Initialization of priority queue Q takes O(V) time. The while loop executes for O(V), where for each execution the block inside the loop takes O(V) times. Hence the total running time is $O(V^2)$.

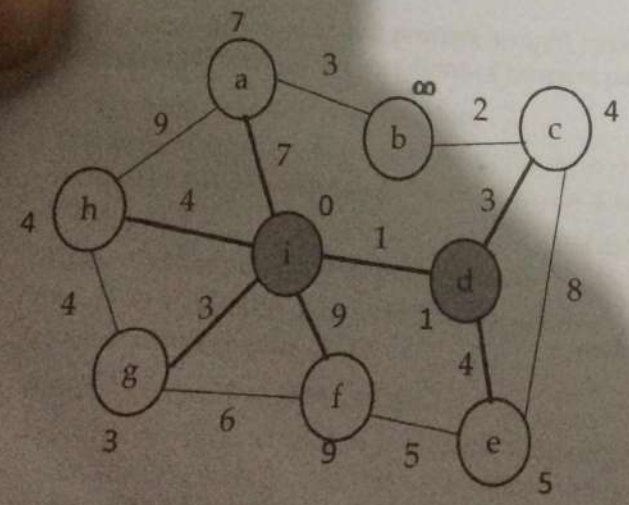
Example: Find the shortest paths from the source node i to all other vertices using Dijkstra's algorithm.

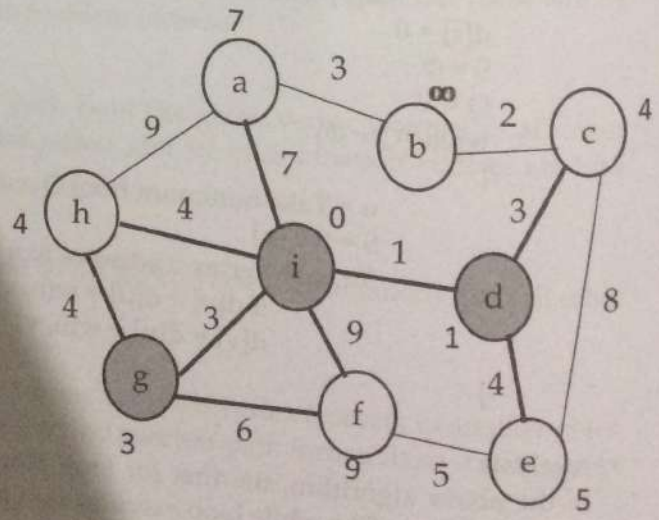


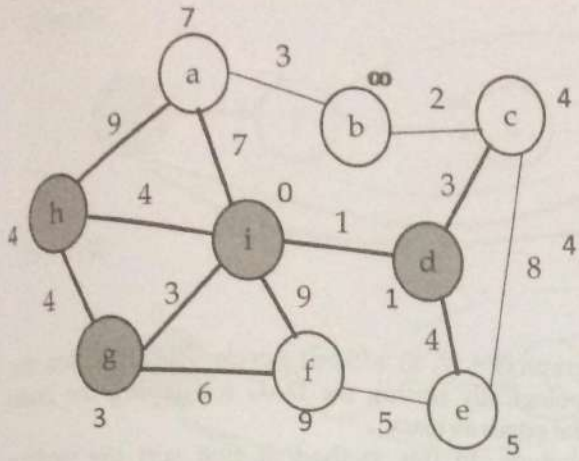
Solution:

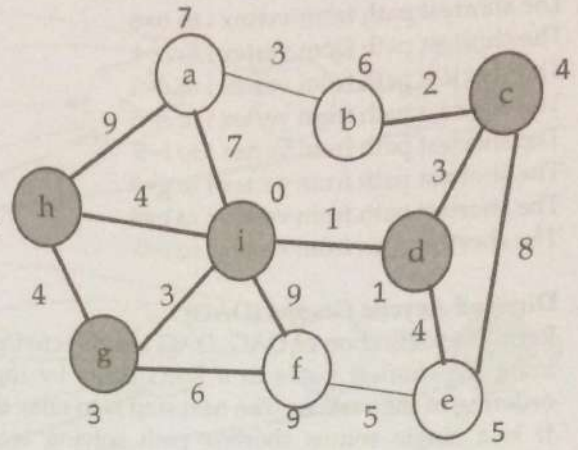


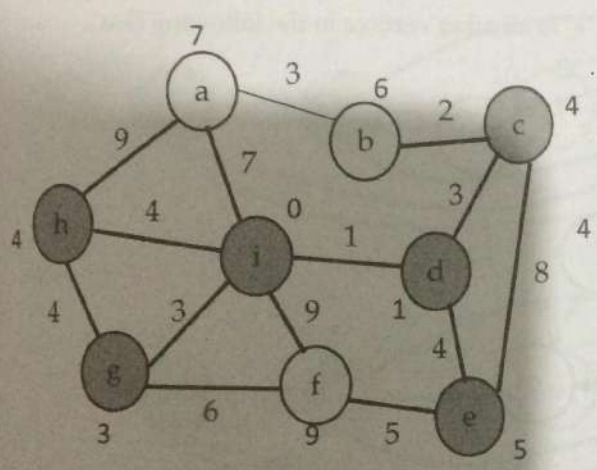


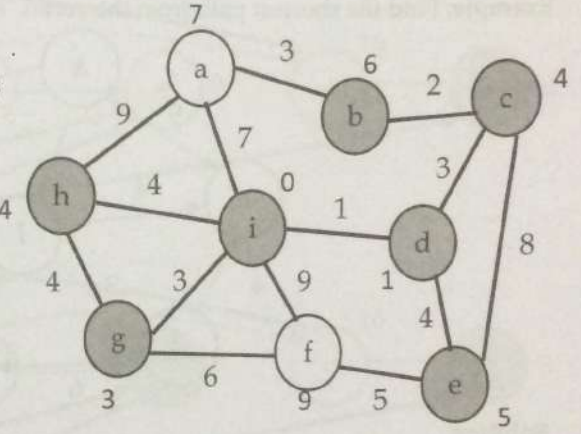


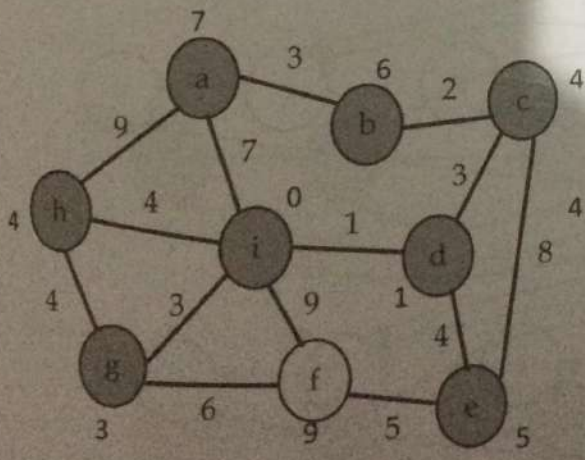


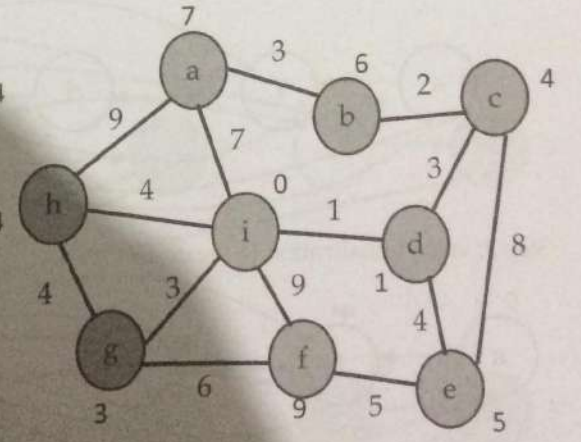

7
a
3
∞
b
2
c
4
9
7
h
4
0
3
4
i
1
d
8
4
3
9
1
g
4
f
6
e
3
9
5
5


7
a
3
6
b
2
c
4
9
7
h
4
0
3
4
i
1
d
8
4
3
9
1
g
4
f
6
e
3
9
5
5


7
a
3
6
b
2
c
4
9
7
h
4
0
3
4
i
1
d
8
4
3
9
1
g
4
f
6
e
3
9
5
5


7
a
3
6
b
2
c
4
9
7
h
4
0
3
4
i
1
d
8
4
3
9
1
g
4
f
6
e
3
9
5
5


7
a
3
6
b
2
c
4
9
7
h
4
0
3
4
i
1
d
8
4
3
9
1
g
4
f
6
e
3
9
5
5


7
a
3
6
b
2
c
4
9
7
h
4
0
3
4
i
1
d
8
4
3
9
1
g
4
f
6
e
3
9
5
5

Thus, the shortest path from vertex i to a=7
The shortest path from vertex i to b=6
The shortest path from vertex i to c=4
The shortest path from vertex i to d=1
The shortest path from vertex i to e=5
The shortest path from vertex i to f=9
The shortest path from vertex i to g=3
The shortest path from vertex i to h=4
The shortest path from vertex i to i=0

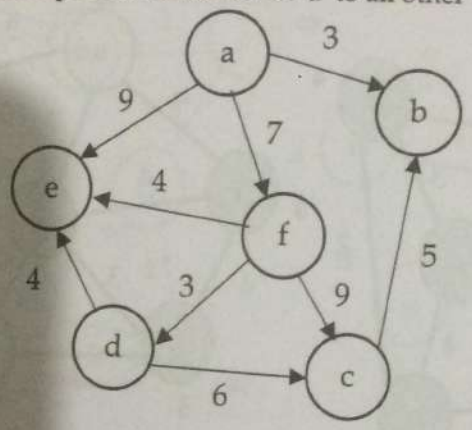
Directed Acyclic Graphs (DAG)

Recall the definition of DAG, DAG is a directed graph G = (V, E) without a cycle. The algorithm that finds the shortest paths in a DAG starts by topologically sorting the DAG for getting the linear ordering of the vertices. The next step is to relax the edges as usual.

It is a single source shortest path solving technique. In this method at first sort the vertices topologically then perform operation same as that of Dijkstra's algorithm.

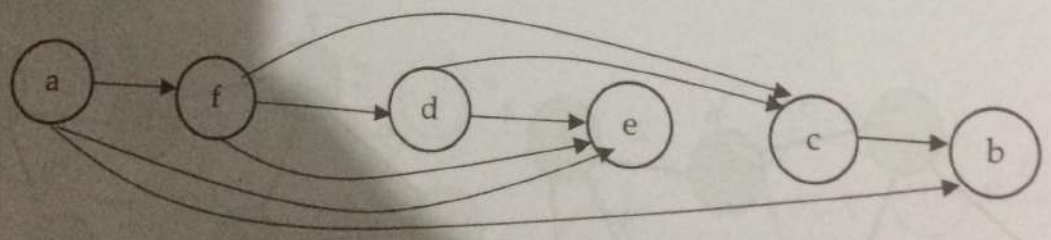
Thus, DAG=Topological sort of vertices + Dijkstra's algorithm

Example: Find the shortest path from the vertex 'a' to all other vertices in the following DAG.

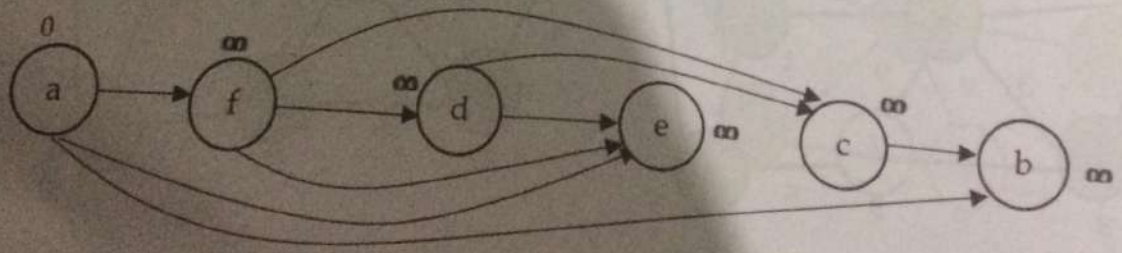


Solution

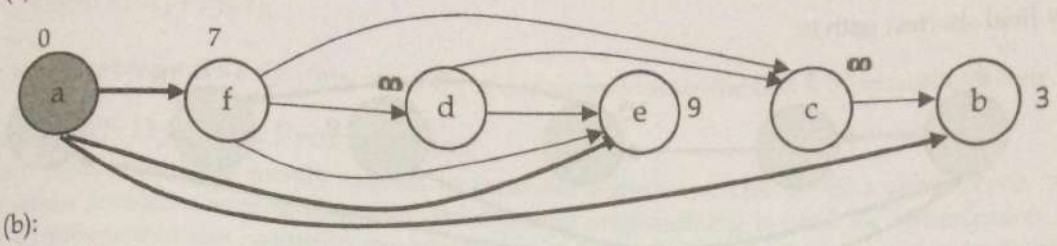
Step 1: Topological sort



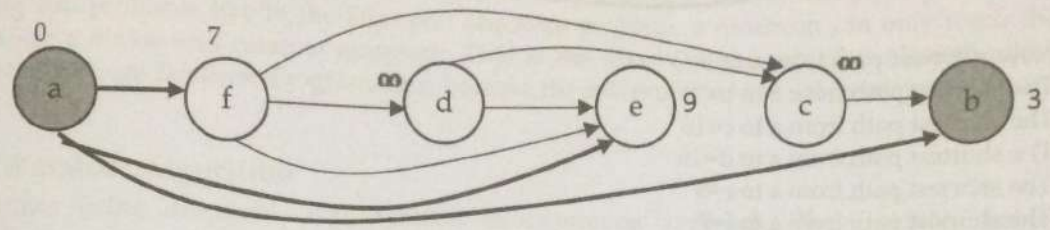
Step 2: value initialization to the nodes



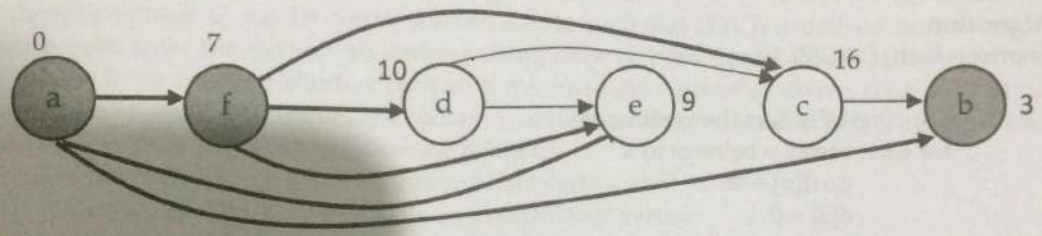
From (a):



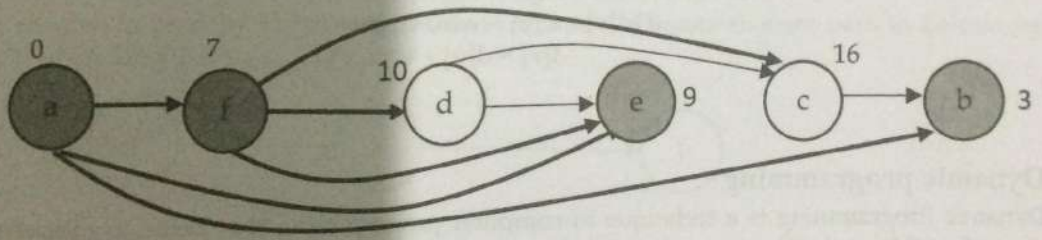
From (b):



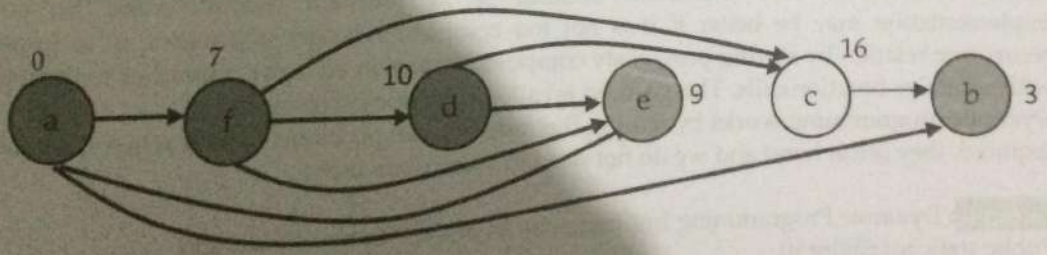
From (f):



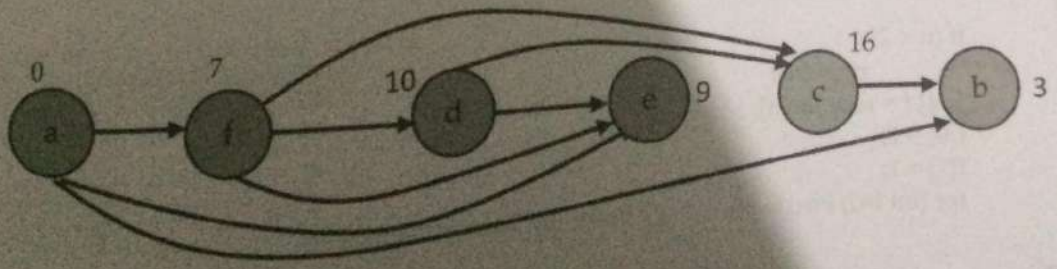
From (e):



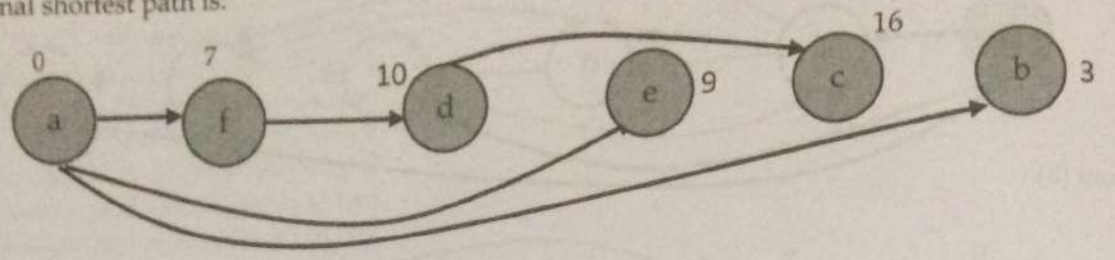
From (d):



From (c):



Now final shortest path is:



Now, shortest path from a to b=3
The shortest path from a to b=3
The shortest path from a to c=16
The shortest path from a to d=10
The shortest path from a to e=9
The shortest path from a to f=7
The shortest path from a to a=0

Algorithm

```
Shortest_Path_DAG(G, w, s)
{
        Topologically Sort the vertices of G
        for each vertex v belongs to V
                do d[v] = Φ
                d[s] = 0
                for each vertex u, taken in topologically sorted order
                        for each vertex v adjacent to u
                                if (d[v] > d[u] + w(u, v)) then,
                                        d[v] = d[u] + w(u, v)
}
```

Dynamic programming

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping sub problems and optimal substructure property. In most cases, recursion is very inefficient because of its frequent function calls. So an iterative implementation may be better if it is not too complex. Another alternative is to implement the recurrence relation by storing previously computed values in an array instead of re-computing them with recursive function calls. This method is called dynamic programming.

Dynamic programming works by storing the result of sub problems so that when their solutions are required, they are at hand and we do not need to recalculate them.

Example: Dynamic Programming Implementation of the Fibonacci Function

```
Public static int fib(int n)
{
        if (n < 2)
                return n;
        int[] f = new int[n];
        f[0] = 0;
        f[1] = 1;
        for (int i=2; i<n; i++)        // Store the Fibonacci numbers
```

```
        f[i] = f[i-1] + f[i-2];
    return f[n-1] + f[n-2];
}
```

This implementation uses a dynamic array f[ ] of n integers to store the first n Fibonacci numbers.

Recursion VS Dynamic Programming

Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization. But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping sub problems like in the Fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach. That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the sub problems are not overlapping in any way.

Floyd's Warshall Algorithm

The algorithm being discussed uses dynamic programming approach. The algorithm being presented here works even if some of the edges have negative weights. Consider a weighted graph G = (V, E) and denote the weight of edge connecting vertices i and j by w_{ij}. Let W be the adjacency matrix for the given graph G. Let D^k denote an nXn matrix such that $D^k(i, j)$ is defined as the weight of the shortest path from the vertex i to vertex j using only vertices from 1,2,...,k as intermediate vertices in the ath. If we consider shortest path with intermediate vertices as above, then computing the path contains two cases. $D^k(i, j)$ does not contain k as intermediate vertex and $D^k(i, j)$ contains k as intermediate vertex. Then we have the following relations

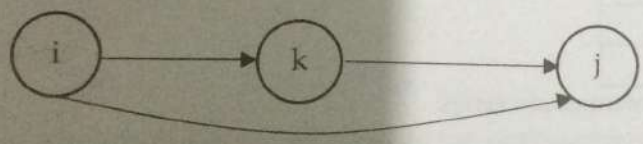
$D^k(i, j) = D^{k-1}(i, j)$, when k is not an intermediate vertex, and

$D^k(i, j) = D^{k-1}(i, k) + D^{k-1}(k, j)$, when k is an intermediate vertex

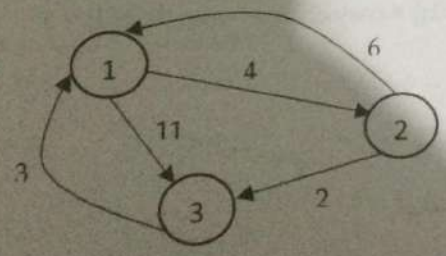
So from above relations we obtain;

$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{k-1}(k, j)\}$

The above relation is used by Floyd's algorithm to compute all pairs shortest path in bottom up manner for finding $D^1, D^2, D^3, \ldots\ldots\ldots D^n$.



Example: Find shortest path from every vertex to other vertices.



Solution: Adjacency matrix of given graph is;

W or D^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Case 1: When vertex (1) as intermediate vertex:

$D^1(1, 1) = 0$

$D^1(1, 2) = \min\{D^0(1, 2), D^0(1, 1) + D^0(1, 2)\}$ = unchanged= min {4, 0+4} = 4

$D^1(1, 3) = \min\{D^0(1, 3), D^0(1, 1) + D^0(1, 3)\}$ = unchanged=min {11, 0+11} = 11

$D^1(2, 1) = \min\{D^0(2, 1), D^0(2, 1) + D^0(1, 1)\}$ =unchanged= min {6, 6+0} = 6

$D^1(2, 2) = 0$

$D^1(2, 3) = \min\{D^0(2, 3), D^0(2, 1) + D^0(1, 3)\}$ = may change=min {2, 6+11} = 2

$D^1(3, 1) = \min\{D^0(3, 1), D^0(3, 1) + D^0(1, 1)\}$ = unchanged= min {3, 3+0} = 3

$D^1(3, 2) = \min\{D^0(3, 2), D^0(3, 1) + D^0(1, 2)\}$ =may change= min {∞, 3+4} = 7

$D^1(3, 3) = 0$

Thus adjacency matrix can be modified as

D^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

Case 2: When vertex (2) as intermediate vertex:

$D^2(1, 1) = 0$

$D^2(1, 2)$ = unchanged= 4

$D^2(1, 3) = \min\{D^1(1, 3), D^1(1, 2) + D^1(2, 3)\}$ = may change=min {11, 4+2} = 6

$D^2(2, 1)$ = unchanged= 6

$D^2(2, 2) = 0$

$D^2(2, 3)$ = unchanged = 2

$D^2(3, 1) = \min\{D^1(3, 1), D^1(3, 2) + D^1(2, 1)\}$ = may change= min {3, 7+6} = 3

$D^2(3, 2)$ = unchanged = 7

$D^2(3, 3) = 0$

Thus adjacency matrix can be modified as

D^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

Case 3: When vertex (3) as intermediate vertex:

$D^3(1, 1) = 0$

$D^3(1, 2) = \min\{D^2(1, 2), D^2(1, 3) + D^2(3, 2)\}$ = may change= min {4, 6+7} = 4

$D^3(1, 3)$ = unchanged = 6

$D^3(2, 1) = \min\{D^2(2, 1), D^2(2, 3) + D^2(3, 1)\}$ = may change= min {6, 2+3} = 5

$D^3(2, 2) = 0$

$D^3(2, 3)$ = unchanged = 2

$D^3(3, 1)$ = unchanged = 3

$D^3(3, 2)$ = unchanged = 7

$D^3(3, 3) = 0$

Thus adjacency matrix can be modified as

D^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Thus, the shortest path form vertex 1 to 1=0

The shortest path form vertex 1 to 2=4

The shortest path form vertex 1 to 3=6
The shortest path form vertex 2 to 1=5
The shortest path form vertex 2 to 2=0
The shortest path form vertex 2 to 3=2
The shortest path form vertex 3 to 1=3
The shortest path form vertex 3 to 2=7
The shortest path form vertex 3 to 3=0

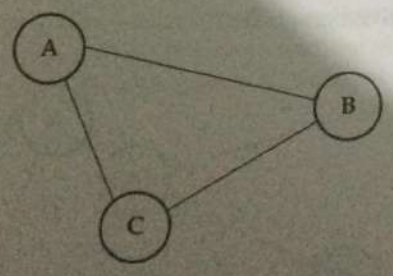
Algorithm: let W be a matrix that contain weight of each edges of given graph G, n be number of nodes and D be the adjacency matrix of given graph.

```
FloydWarsal_ASP(W, D, n)
{
        for(i=1; i<=n; i++)
        {
                for(j=1; j<=n; j++)
                {
                        D[i][j]=W[i][j];   //Original D⁰ matrix
                }
        }
        for(k=1; k<=n; k++)
        {
                for(i=1; i<=n; i++)
                {
                        for(j=1; j<=n; j++)
                        {
                                if(D[i][j]>D[i][k]+D[k][j]) then
                                        Set, D[i][j]=D[i][k]+D[k][j]
                        }
                }
        }
}
```

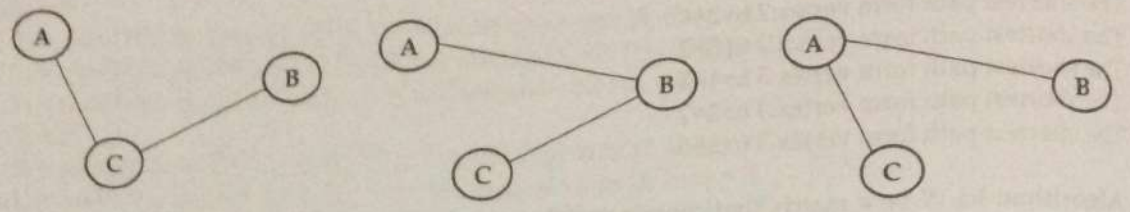
Spanning tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected. Simply, the all possible trees constructed from given graph are called spanning tree of such a graph.

Example: Find spanning trees of given graph below:



Solution: Possible spanning trees of given graph are:



Properties of Spanning Tree

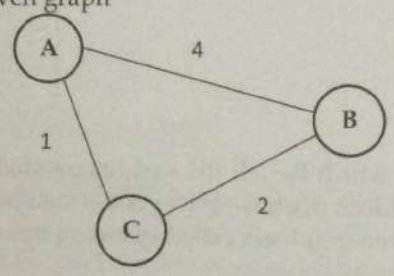
We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

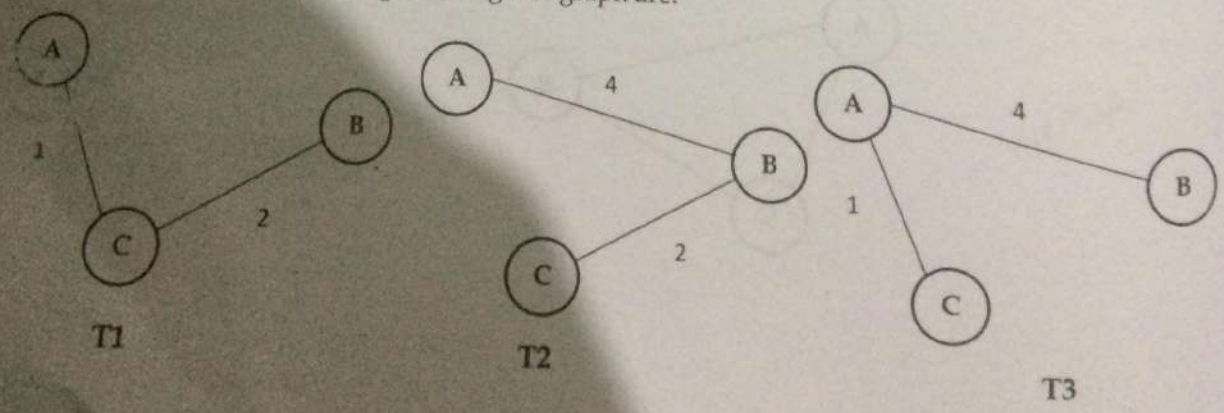
Minimum spanning tree

A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible total weights of its edges out of all possible spanning trees. In other words in a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Example: Find MST of given graph



Solution: The possible spanning trees of given graph are:



Total weight of T1=3
Total weight of T2=6
Total weight of T3=5
Since T1 has minimum weight out of all possible spanning trees of given graph hence, tree T1 acts as minimum spanning tree of given graph.

Kruskal's algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which:

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

It is the procedure for producing a minimum spanning tree of a given weighted graph that successively adds edges of least weight that are not already in the tree such that no edges produce a simple circuit when it is added.

Algorithm

1. Start
2. Sort all the edges from low weight to high
3. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
4. Keep adding edges until we reach all vertices.
5. Stop

Pseudo code

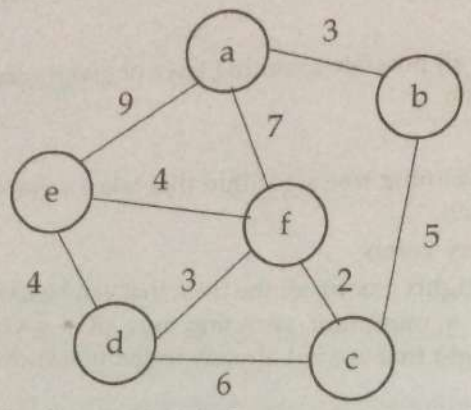
Let G be a weighted connected undirected graph with n vertices)

```
KruskalMST(G)
{
        T = {V}  // forest of n nodes
        E = set of edges sorted in non-decreasing order of weight
        while(|T| < n-1 and E! =Φ)
        {
                    •   Select (u, v) from E in order
                    •   Remove (u, v) from E
                    •   If ((u, v) does not create a cycle in T))
                                T = T U {(u, v)}
        }
}
```

Analysis

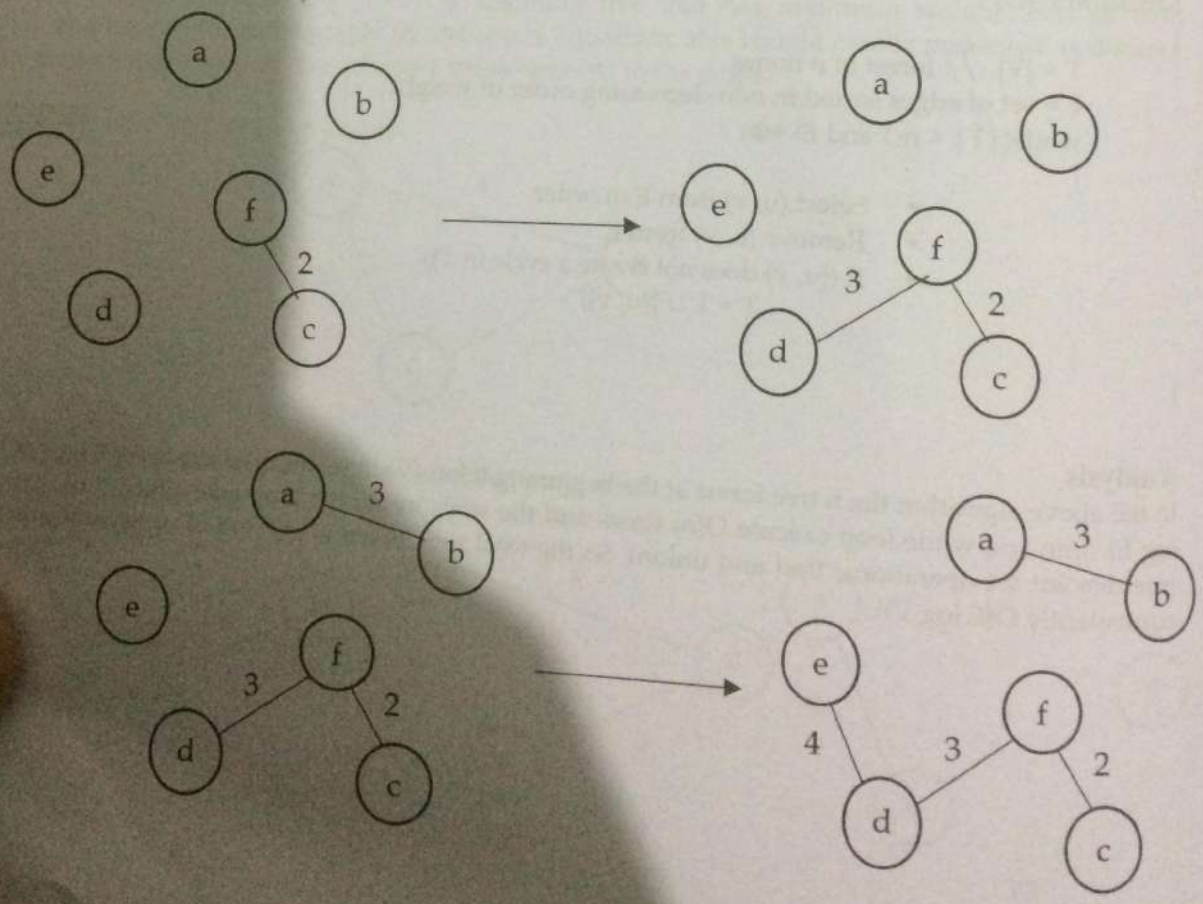
In the above algorithm the n tree forest at the beginning takes (V) time, the creation of set S takes O(E log E) time and while loop execute O(n) times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is O(E log E) or asymptotically equivalently O(E log V)!.

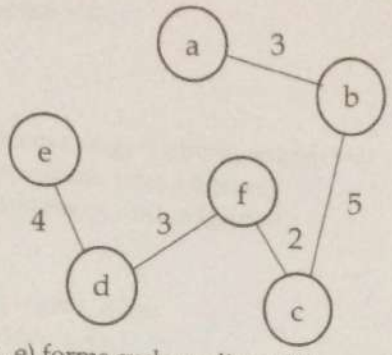
Example: Find minimum spanning tree of given graph by using Kruskal's algorithm.



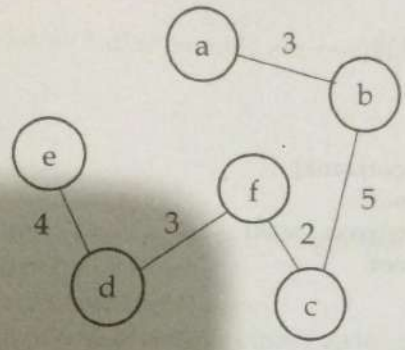
Solution: Edges in sorted order are given below:

Edges	Cost	Action
(f, c)	2	Accept
(f, d)	3	Accept
(a, b)	3	Accept
(d, e)	4	Accept
(b, c)	5	Accept
(d, c)	6	Reject
(a, f)	7	Reject
(a, e)	9	Reject





The edges (d, c), (a, f) and (a, e) forms cycle so discard these edges from the tree. Thus the minimum spanning tree of weight 17 is;



Complete C program for Kruskal's algorithm to find Minimum Spanning Tree

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Edge
{
        int src;
        int dest;
        int weight;
};

struct Graph
{
        int V, E;
        struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
        struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
        graph→V = V;
        graph→E = E;
        graph→edge = (struct Edge*) malloc( graph→E * sizeof( struct Edge ) );
```

```
        return graph;
}

struct subset
{
        int parent;
        int rank;
};

int find(struct subset subsets[], int i)
{
        if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
        return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y)
{
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);
        if (subsets[xroot].rank < subsets[yroot].rank)
                subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
                subsets[yroot].parent = xroot;

        else
        {
                subsets[yroot].parent = xroot;
                subsets[xroot].rank++;
        }
}

int myComp(const void* a, const void* b)
{
        struct Edge* a1 = (struct Edge*)a;
        struct Edge* b1 = (struct Edge*)b;
        return a1→weight > b1→weight;
}

void KruskalMST(struct Graph* graph)
{
        int V = graph→V;
        struct Edge result[V];
        int e = 0;
        int i = 0;
        qsort(graph→edge, graph→E, sizeof(graph→edge[0]), myComp);
        struct subset *subsets =(struct subset*) malloc( V * sizeof(struct subset) );
        for (int v = 0; v < V; ++v)
        {
                subsets[v].parent = v;
```

```
            subsets[v].rank = 0;
        }
        while (e < V - 1)
        {
            struct Edge next_edge = graph→edge[i++];
            int x = find(subsets, next_edge.src);
            int y = find(subsets, next_edge.dest);
            if (x != y)
            {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
    {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
    }
    return;
}
```

Prim's algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which:

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

The idea behind this algorithm is just take any arbitrary vertex and choose the edge with minimum weight incident on the chosen vertex. Add the vertex and continue the above process until all the vertices are not added to the list. Remember the cycle must be avoided.

Algorithm

1. Start
2. Initialize the minimum spanning tree with a vertex chosen at random.
3. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
4. Keep repeating step 3 until we get a minimum spanning tree
5. Stop

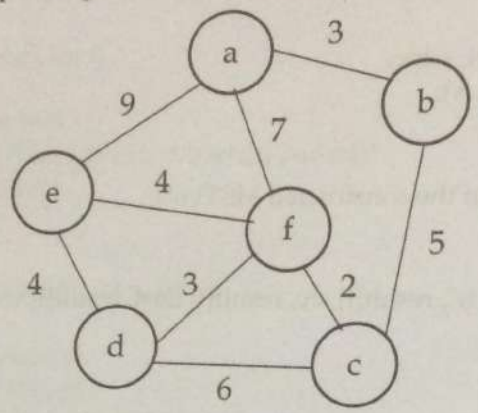
Pseudo code

```
PrimMST(G)
{
    T = Φ;              // T is a set of edges of MST
    S = {s}  ;          //s is randomly chosen vertex and S is set of vertices
    while (S! = V)
    {
        e = (u, v) an edge of minimum weight incident to vertices in T and not forming
        simple circuit in T if added to T i.e. u ∈ S and v ∈ V-S
        T = T U {(u, v)};
        S = S U {v};
    }
}
```

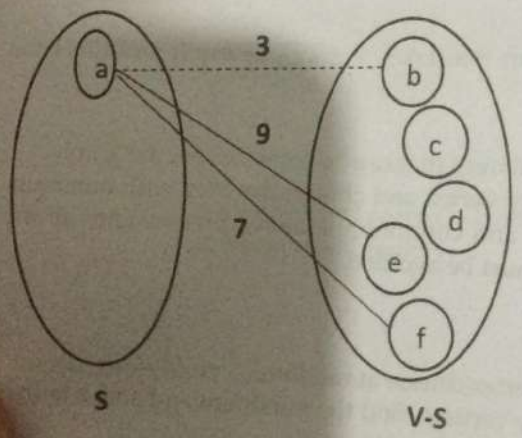
Analysis

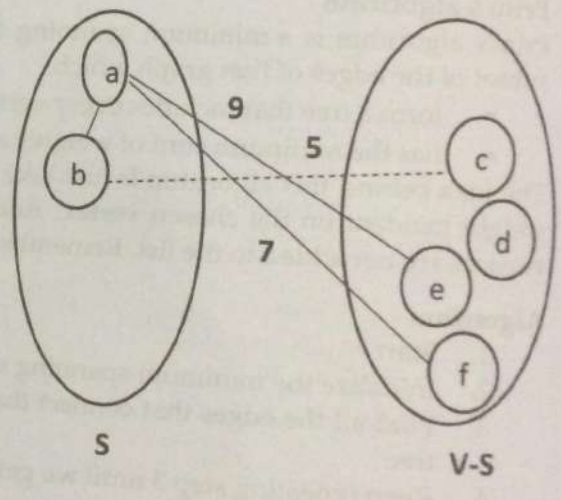
In the above algorithm while loop execute O(V). The edge of minimum weight incident on a vertex can be found in O(E), so the total time is O(EV). We can improve the performance of the above algorithm by choosing better data structures as priority queue and normally it will be seen that the running time of prim's algorithm is O(E log V)!.

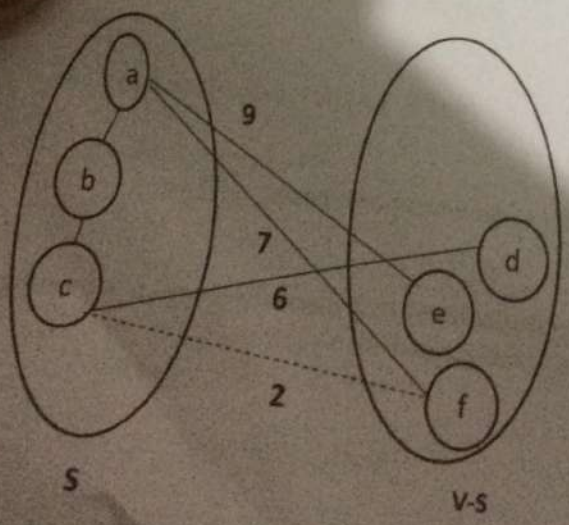
Example: find minimum spanning tree of given graph by using Prim's algorithm

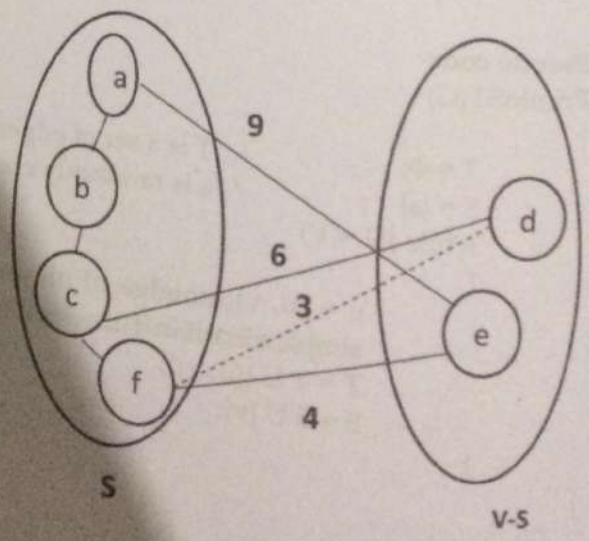


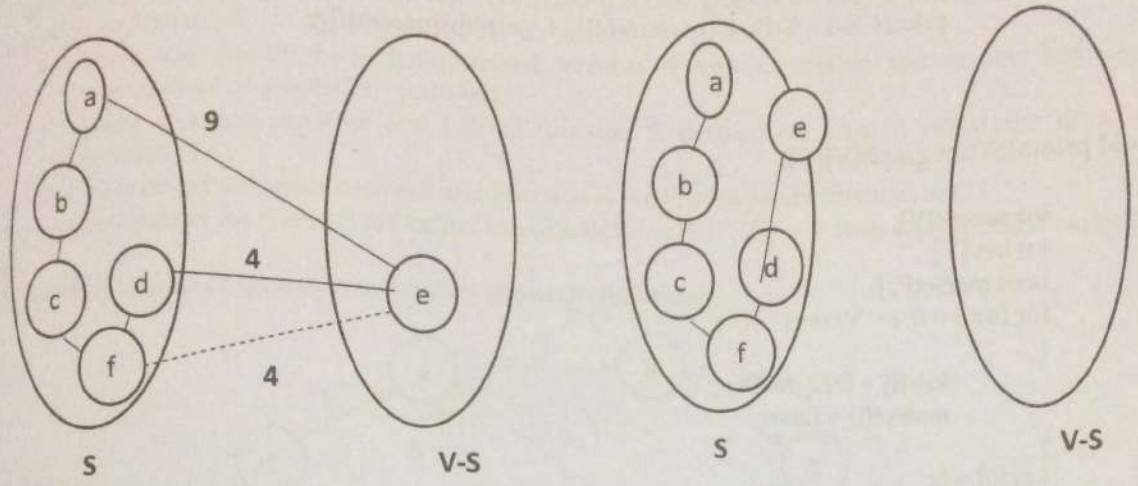
Solution:



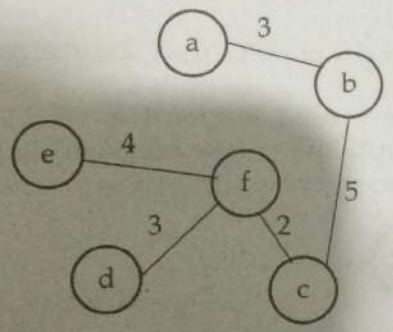








Thus final MST given by Prim's algorithm is given below;



Thus the minimum spanning tree of weight 17 is shown in fig above.

Complete C program to find MST from given graph

Complete C program for Prim's Minimum Spanning Tree (MST) algorithm

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minKey(int key[ ], bool mstSet[])
{
        int min =INT_MAX, min_index;    //INT_MAX gives maximum value for int
        for (int v = 0; v < V; v++)
        {
                if (mstSet[v] == false && key[v] < min)
                        min = key[v], min_index = v;
        }
        return min_index;
}
int printMST(int parent[ ], int n, int graph[V][V])
{
        printf("Edge   Weight\n");
```

```
        for (int i = 1; i < V; i++)
        {
                printf("%d - %d%d \n", parent[i], i, graph[i][parent[i]]);
        }
}

void primMST(int graph[V][V])
{
        int parent[V];
        int key[V];
        bool mstSet[V];
        for (int i = 0; i < V; i++)
        {
                key[i] = INT_MAX;
                mstSet[i] = false;
        }
        key[0] = 0;
        parent[0] = -1;
        for (int count = 0; count < V-1; count++)
        {
                int u = minKey(key, mstSet);
                mstSet[u] = true;
                for (int v = 0; v < V; v++)
                {
                        if (graph[u][v] && mstSet[v] == false && graph[u][v] <  key[v])
                        {
                        parent[v]  = u;
                        }
                        key[v] = graph[u][v];
                }
        }
    printMST(parent, V, graph);
}
int main()
{
        int graph[V][V] = {{0, 2, 0, 6, 0},
                           {2, 0, 3, 8, 5},
                           {0, 3, 0, 0, 7},
                           {6, 8, 0, 0, 9},
                           {0, 5, 7, 9, 0},
                };
                primMST(graph);
                return 0;
}
```

Network Flow Problems

Some real life problems like those involving the flow of liquids through pipe, current through wires, and delivery of goods can be modeling by using flow of networks.

Transport network

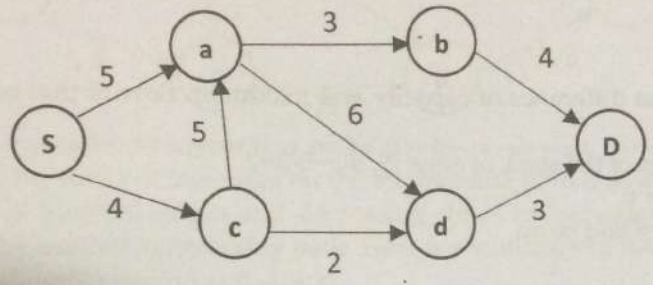
A flow network or transport network is a connected directed graph G = (V, E) that does not contain any loops such that,

- There are exactly two distinguished vertices S and D, called the source and sink (destination) of graph G respectively
- There is a non-negative real valued function K defined on edge E called the capacity function of G.

Then (G, K) is called transport network and function K is called capacity function of G.

The vertices distinct for S and D are called intermediate vertices. If e ∈ E then K(e) is called capacity of e.

Example: Example of transport network is shown in fig below;

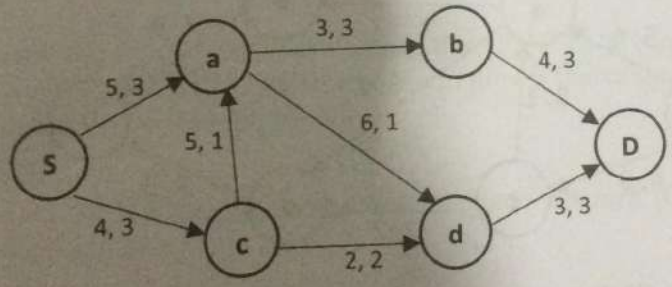


Flow

Let (G, K) be a transport network. Then a flow in G is a non-negative real valued function F defined on edge E such that,

$0 \le F(e) \le K(e)$ for each edge e belongs to E. Where, F(e)=flow to the edge e and K(e)=maximum capacity to the edge e.

Example: Following transport network show that the capacity and flow in each of the edges.



From the above figure the flow-in of vertex 'a' is F(S, a) + F (c, a) = 3+1=4

The flow out of vertex 'a' is F(a, b)+F(a, d)=3+1=4

The flow in S=0

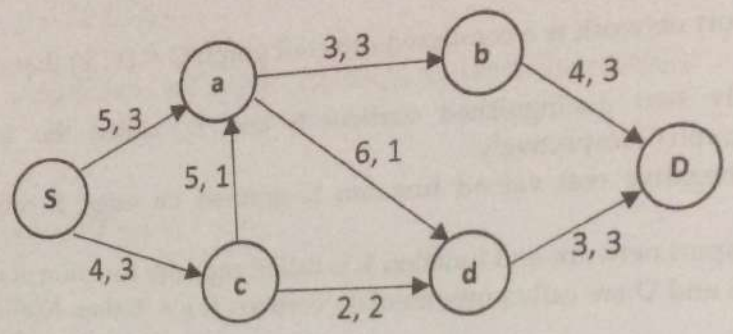
Flow in of vertex D=F(b, D) + F(d, D)=3+3=6 and so on

Note: Flow out of source vertex = flow in of sink vertex

Saturated and Unsaturated edges

The edges for which the flow and capacity are equal are called saturated edges. Otherwise they are called unsaturated edges.

Example:



Here edges (a, b), (c, d) and (d, D) are called saturated edges. And remaining edges are called unsaturated edges.

Slack of edge

The slack of edge 'e' is the difference of capacity and maximum flow of that edge. The slack of each saturated edge is zero.

Example: In the above figure the slack of edge (S, a) = 5-3=2

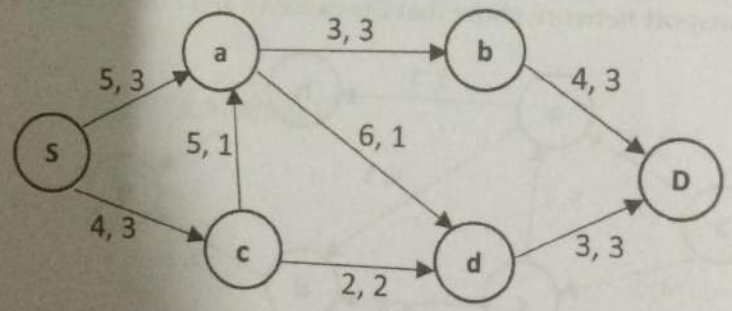
Slack of edge (a, b) = 3-3 = 0

Slack of edge (a, d) = 6-1=5 and so on

Cut

Let (G, K) is a transport network with source S and sink D. let X is subset of V and Y= V-X where X contains at least source S and Y contains at least sink D. Then the edge list (X, Y) are called cut edges of given network flow.

Example:



Let X={S, a, d} and Y= {b, c, D} then

(X, Y)= {(S, c), (a, b), (d, D)} is called cut edge of given transport network.

Minimum cut

A cut edge (X, Y) is said to be minimum cut edge of transport network (G, K) if there is no any cut edge (W, Z) that is less than (X, Y). I.e. K(W, Z) < K(X, Y) is false.

Example: In the above figure possible cut edges and their capacity are,

Case 1: let X={S} and Y= {a, b, c, d, D} then

(X, Y)= {(S, a), (S, c)} =5+4=9

Case 2: let X={S, a} and Y= {b, c, d, D} then

(X, Y)= {(S, c)} =4

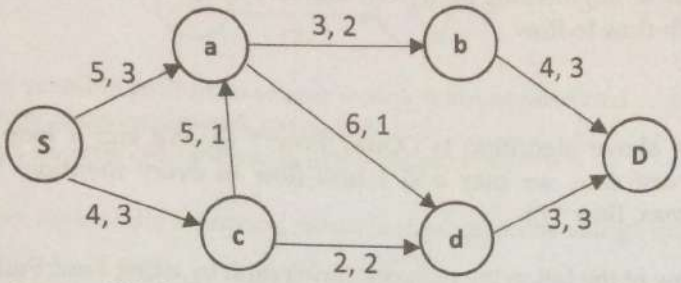
Case 3: let X={S} and Y= {a, b, c, d, D} then

(X, Y)= {(S, a), (S, c)} =5+4=9

F-augmenting path or flow augmenting path

A flow-augmenting path from source *s* to sink *t* is a sequence of edges from *s* to *t* such that, for each edge in this path, the flow F(e) < K(e) on forward edges and F(e) > 0 on backward edges. It means that such a path is not optimally used yet, and it can transfer more units than it is currently transferring. If the flow for at least one edge of the path reaches its capacity, then obviously the flow cannot be augmented.

Example:



In the above figure the possible f-augmenting paths are: {S→a→b→D}, {S→c→a→b→D}

If we reach the sink t, the flows of the edges on the augmenting path that was just found are updated by increasing flows of forward edges and decreasing flows of backward edges, and the process restarts in the quest for another augmenting path. Here is a summary of the algorithm.

```
augmentPath (network with source s and sink t)
        for each edge e in the path from s to t
                if forward(e)
                        f(e) += slack(t);
                else
                        f(e) -= slack(t);
```

Maximum flow

A flow F in a network (G, K) is called a maximum flow if |F'|≤ |F, for every flow F' in (G, K). Simply a flow F in a network (G, K) is called maximum flow if the value of F is the largest possible value of any flow other possible flows in (G, K).

Ford Fulkerson algorithm

This algorithm is used to find the maximum flow of given network (G, K). Let a network flow diagram with source s and sink t.

```
1. Start
2. set flow of all edges and vertices to 0;
   Label = (null, ∞);
   Labeled = {s};
3. while labeled is not empty // while not stuck;
   3.1.  Detach a vertex v from labeled;
   3.2.  for all unlabeled vertices u adjacent to v
         3.2.1. if forward(edge(vu)) and slack(edge(vu)) > 0
                     Label (u) = (v+, min (slack (v), slack (edge (vu))))
         3.2.2. else if backward(edge(vu)) and f(edge(uv)) > 0
                     Label (u) = (v–, min (slack (v), f (edge (uv))));
         3.2.3. if u got labeled
           3.2.3.1.  if u == t
                          augmentPath (network);
                               Labeled = {s};     // look for another path;

           3.2.3.2.  else
```

Include u in labeled;

4. Stop

Simple idea behind Ford-Fulkerson Algorithm

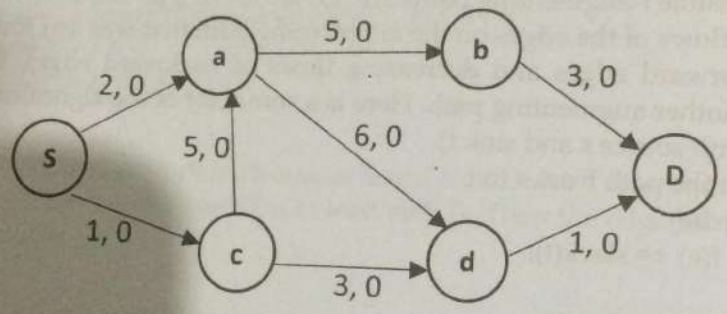
The following is simple idea of Ford-Fulkerson algorithm

1. Start with initial flow as 0
2. While there is 'a' augmenting path from source to sink.
   Add this path-flow to flow
3. Return flow.

Time Complexity

Time complexity of the above algorithm is O(max_flow * E). We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max_flow * E).

Example 1: Find max flow of the following network flow graph by using Ford Fulkerson Algorithm.



Solution: At first listing f-augmenting paths as

{S→a→b→D},{S→c→d→D},{S→a→d→D},{S→a→c→d→D}, {S→c→d→a→b→D}

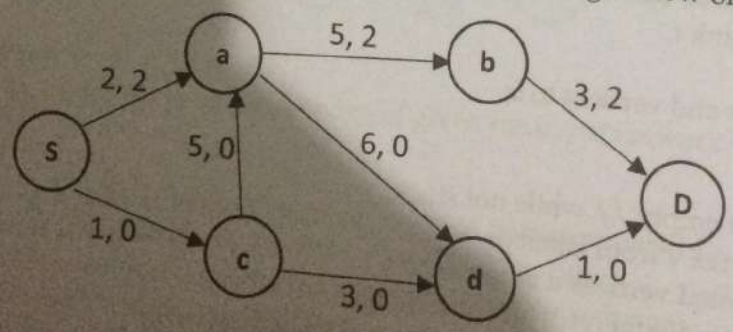
Step 1: In the f-augmenting path {S→a→b→D},

Slack of edge (S, a)=2-0=2 (minimum)

Slack of edge (a, b)=5-0=5

Slack of edge (b, D)=3-0=3

Since the minimum slick is 2 hence add 2 to every edge's flow of the path {S→a→b→D},



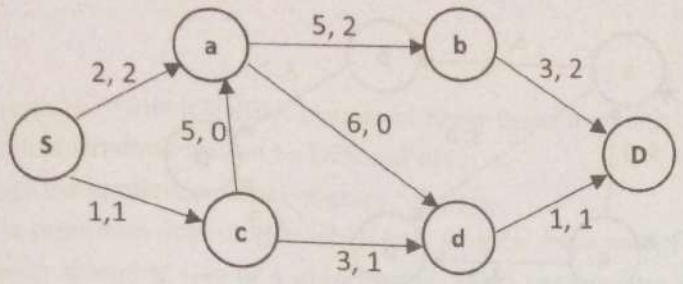
Step 2: In the f-augmenting path {S→c→d→D},

Slack of edge (S, c) =1-0=1 (minimum)

Slack of edge (c, d) =3-0=3

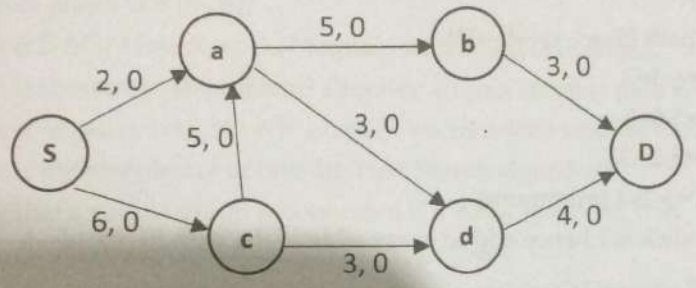
Slack of edge (d, D) =1-0=1

Since the minimum slick is 1 hence add 1 to every edge of the path {S→c→d→D},



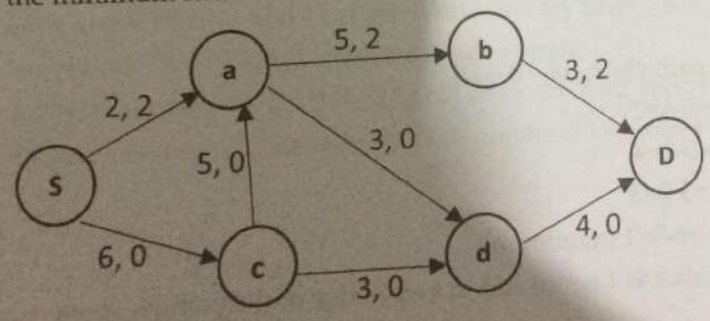
Now there is no any possible path form source to sink without saturated edge
Hence maximum flow of given network graph is 2+1=3
Hence flow out from source=2+1=3=flow in to the sink=2+1=3.

Example 2: Find max flow of the following network flow graph by using Ford Fulkerson Algorithm.

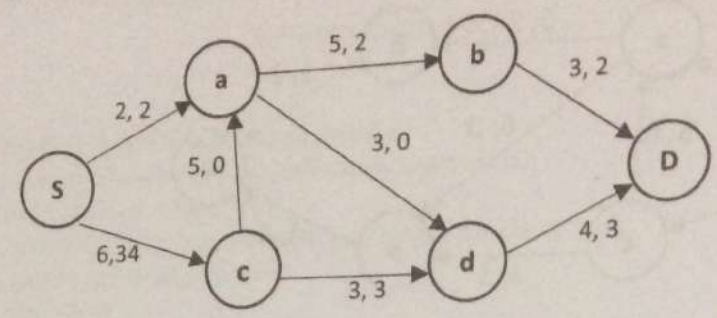


Solution: At first listing f-augmenting paths as
{S→a→b→D},{S→c→d→D},{S→a→d→D},{S→a→c→d→D},{S→c→a→b→D},
,{S→c→a→d→D}, {S→c→d→a→b→D}

Step 1: In the f-augmenting path {S→a→b→D},
Slack of edge (S, a)=2-0=2 (minimum)
Slack of edge (a, b) =5-0=5
Slack of edge (b, D) =3-0=3
Since the minimum slick is 2 hence add 2 to every edge's flow of the path {S→a→b→D},



Step 2: In the f-augmenting path {S→c→d→D},
Slack of edge (S, c) =6-0=6
Slack of edge (c, d) =3-0=3 (minimum)
Slack of edge (d, D) =4-0=4
Since the minimum slick is 3 hence add 3 to every edge of the path {S→c→d→D},



Step 3: In the f-augmenting path {S→a→d→D},

No change

Step 4: In the f-augmenting path {S→a→c→d→D},

No change

Step 5: In the f-augmenting path {S→c→a→b→D},

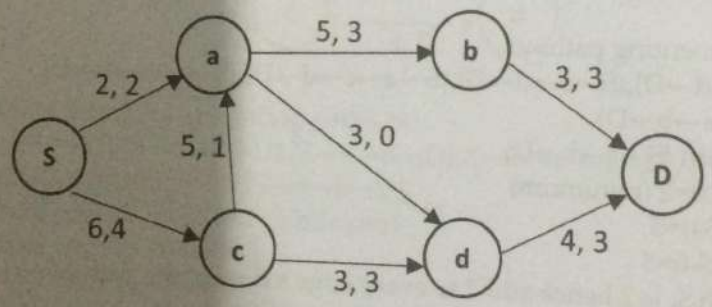
Slack of edge (S, c) =6-3=3

Slack of edge (c, a) =5-0=5

Slack of edge (a, b) =5-2=3

Slack of edge (b, D) =3-2=1 (minimum)

Since the minimum slick is 1 hence add to every edge of the path {S→c→a→b→D},



Step 6: In the f-augmenting path {S→c→a→d→D},

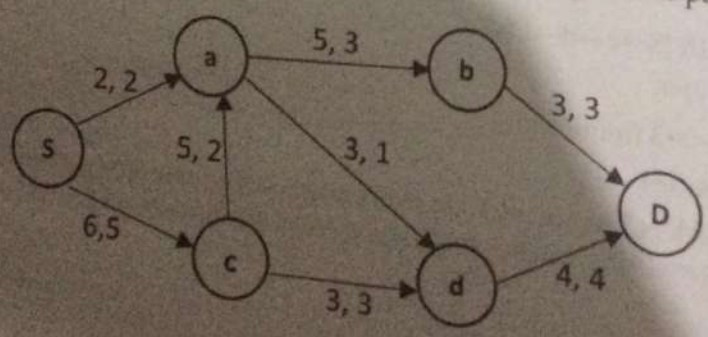
Slack of edge (S, c) =6-4=2

Slack of edge (c, a) =5-1=4

Slack of edge (a, d) =3-0=3

Slack of edge (d, D) =4-3=1 (minimum)

Since the minimum slick is 1 hence add to every edge of the path {S→c→a→d→D},



Exercise

1. What is graph? How it is differ from tree? Show that a tree with n vertices has n – 1 edge.
2. Which data structure is used by DFS and BFS?
3. What are the application areas of graph? Explain.
4. What is minimum spanning tree? How it is differ from maximum spanning tree? Draw a maximum spanning tree of a graph containing 5 vertices and 9 edges with arbitrary edge cost.
5. What do you mean by representation of graph? Which representation of graph is more suitable to represent complete graph?
6. What is the relationship between the sum of the degrees of all vertices and the number of edges of graph G = (V, E)?
7. What is DAG? How it works? Explain with suitable example.
8. What is shortest path problem? Describe all pair shortest path with suitable example.
9. What is negative weight cycle graph? Describe with suitable example.
10. What is the complexity of breadth First Search algorithm?
11. Show that a simple graph is connected if it has a spanning tree.
12. What do you mean by topological sort? Explain.
13. What do you mean by all pair shortest path problem? Describe Floyd Warshall algorithm.
14. What do you mean by graph traversing? Which data structure is used to efficiently implement the DFS and BFS?
15. How can Dijkstra's Algorithm be applied to undirected graphs? Explain.
16. How can Dijkstra's Algorithm be modified to become an algorithm for finding the shortest path from vertex 'a' to 'b'?
17. What is the application of spanning tree? Draw a minimum spanning tree of a graph containing any 8 vertices and 11 edges with arbitrary edge costs.
18. What is network flow? Explain with suitable example.
19. Write down statement of max flow min cut theorem. And verify with suitable example.
20. What do you mean by all pair shortest path problem? Describe Floyd Warshall algorithm.

•••

Chapter

11

Hashing

It is an efficient searching technique in which key is placed in direct accessible address for rapid search. Hashing provides the direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. This technique uses a hashing function say h which maps the key with the corresponding key address or location.

Hashing is a different approach to searching which calculates the position of the key in the table based on the value of the key. The value of the key is the only indication of the position. When the key is known, the position in the table can be accessed directly, without making any other preliminary tests, as required in a binary search or when searching a tree. This means that the search time is reduced from O(n), as in a sequential search, or from O(log n), as in a binary search, to at least O(1); regardless of the number of elements being searched, the run time is always the same. But this is just an ideal, and in real applications, this ideal can only be approximated.

Hashing is a method and useful technique to implement dictionaries. This method is used to perform searching, insertion and deletion at a faster rate. A function called Hash Function is used to compute and return position of the record instead of searching with comparisons. The data is stored in array called as Hash table. The Hash Function maps keys into positions in a hash table. The mapping of keys to indices of a hash table is known as Hash Function. The major requirement of hash function is to map equal keys to equal indices.

Given a key, the algorithm computes an index that suggests where the entry can be found:

Index = f(key, array_size)

The value of index is determined by 2 steps

- hash = hash_func(key)
- index = hash % array_size

Hash Function

A function that transforms a key into a table index is called a hash function. A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1. Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. Our first hash function, sometimes referred to as the "remainder method," simply takes an item and divides it by the table size, returning the remainder as its hash value **h(item)=item%11**. Table below gives all of the hash values for our example items. Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Table: Simple Hash Function Using Remainders

Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown in Figure below.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Figure: Hash Table with Six Items

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is O(1), since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a constant time search algorithm.

Types of Hash Functions

There are different types of hash functions are used some of them are described below:

Division

A hash function must guarantee that the number it returns is a valid index to one of the table cells. The simplest way to accomplish this is to use division modulo TSize =sizeof(table), as in h(K) = K mod TSize, if K is a number. It is best if TSize is a prime number; otherwise, h(K) = (K mod p) mod TSize for some prime p > TSize can be used. However, nonprime divisors may work equally well as prime divisors provided they do not have prime factors less than 20. The division method is usually the preferred choice for the hash function if very little is known about the keys.

Folding

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43, 65, 55, 46, 01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get 43+56+55+64+01=219 which gives 219 % 11=10.

Mid-Square Function

In the mid-square method, the key is squared and the middle or mid part of the result is used as the address. If the key is a string, it has to be preprocessed to produce a number by using, for instance, folding. In a mid-square hash function, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys. For example, if the key is 3,121 then $(3,121)^2 = 9,740,641$, and for the 1,000-cell table, $h(3,121) = 406$, which is the middle part of $(3,121)^2$. In practice, it is more efficient to choose a power of 2 for the size of the table and extract the middle part of the bit representation of the square of a key. If we assume that the size of the table is 1,024, then, in this example, the binary representation of $(3,121)^2$ is the bit string 100101001010000101100001, with the middle part shown in italics. This middle part, the binary number 0101000010, is equal to 322. This part can easily be extracted by using a mask and a shift operation.

Extraction

In the extraction method, only a part of the key is used to compute the address. For the social security number 123-45-6789, this method might use the first four digits, 1,234; the last four, 6,789; the first two combined with the last two, 1,289; or some other combination. Each time, only a portion of the key is used. If this portion is carefully chosen, it can be sufficient for hashing, provided the omitted portion distinguishes the keys only in an insignificant way. For example, in some university settings, all international students' ID numbers start with 999. Therefore, the first three digits can be safely omitted in a hash function that uses student IDs for computing table positions. Similarly, the starting digits of the ISBN code are the same for all books published by the same publisher (e.g., 0534 for Brooks/Cole Publishing Company). Therefore, they should be excluded from the computation of addresses if a data table contains only books from one publisher.

Hash collision

In computer science, a **collision or clash** is a situation that occurs when two distinct pieces of data have the same hash value, checksum, fingerprint, or cryptographic digest. Collisions are unavoidable whenever members of a very large set (such as all possible person names, or all possible computer files) are mapped to a relatively short bit string. This is merely an instance of the pigeonhole principle. The impact of collisions depends on the application. When hash functions and fingerprints are used to identify similar data, such as homologous DNA sequences or similar audio files, the functions are designed so as to maximize the probability of collision between distinct but similar data.

Collision Resolution

When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. If the hash function is perfect,

collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing. Some popular methods for minimizing collision are:

- Open addressing
  - ✓ Linear probing
  - ✓ Quadratic probing
  - ✓ Double hashing
- Rehashing
- Chaining
- Hashing using buckets etc

Open Addressing

In open addressing, when a data item can't be placed at the index calculated by the hash function, another location in the array is sought. We'll explore three methods of open addressing, which vary in the method used to find the next vacant cell. These methods are **linear probing, quadratic probing,** and **double hashing**.

Linear probing

A hash table in which a collision is resolved by putting the item in the next empty place within the occupied array space is called linear probing. It starts with a location where the collision occurred and does a sequential search through a hash table for the desired empty location. Hence this method searches in straight line, and it is therefore called linear probing. The main disadvantage of this process is clustering problem.

Example Insert keys {89, 18, 49, 58, 69} with the hash function h(x) =x mod 10 using linear probing.

Solution: when x=89

h(89)=89%10=9

Insert key 89 in hash-table in location 9

When x=18

h(18)=18%10=8

Insert key 18 in hash-table in location 8

When x=49

h(49)=49%10=9 (Collision occur)

So insert key 49 in hash-table in next possible vacant location of 9 is 0

When x=58

h(58)=58%10=8 (Collision occur)

Insert key 58 in hash-table in next possible vacant location of 8 is 1 (since 9, 0 already contains values).

When x=69

h(89)=69%10=9 (Collision occur)

Insert key 69 in hash-table in next possible vacant location of 9 is 2 (since 0, 1 already contains values).

0	1	2	3	4	5	6	7	8	9
49	58	69	None	None	None	None	None	18	89

Complete C program for linear probing

Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem take place in a linear probing. When collision occur then the quadratic probing works as follows:

(Hash value + 1^2)% table size

If there is again collision occurs then there exist rehashing.

(Hash value + 2^2)%table size

If there is again collision occurs then there exist rehashing.

(Hash value = 3^2)% table size

In general in i^{th} collision

$h_i(x)$=(hash value +i^2)% table size

Example: Insert keys {89, 18, 49, 58, and 69} with the hash-table size 10 using quadratic probing.

Solution: when x=89

h(89)=89%10=9

Insert key 89 in hash-table in location 9

When x=18

h(18)=18%10=8

Insert key 18 in hash-table in location 8

When x=49

h(49)=49%10=9 (Collision occur)

So use following hash function,

h1(49)=(49 + 1)%10=0

Hence insert key 49 in hash-table in location 0

When x=58

h(58)=58%10=8 (Collision occur)

So use following hash function,

h1(58)=(58 + 1)%10=9

Again collision occur use again the following hash function,

h2(58)=(58+ 22)%10=2

Insert key 58 in hash-table in location 2

When x=69

h(89)=69%10=9 (Collision occur)

So use following hash function,

h1(69)=(69 + 1)%10=0

Again collision occur use again the following hash function,

h2(69)=(69+ 22)%10=3

Insert key 69 in hash-table in location 3

0	1	2	3	4	5	6	7	8	9
49	None	58	69	None	None	None	None	18	89

Double hashing

Double hashing is a collision resolution method that eliminates the primary clustering problem take place in a linear probing. When collision occur then the double hashing define new hash function as follows:

$h_2(x) = R - (x \bmod R)$

Where, R is a prime number smaller than hash table size.

The new element can be inserted in the position by using following function,

Position = (original hash value + $i*h_2(x)$) % Table size

Example: Insert keys {89, 18, 49, 58, and 69} with the hash-table size 10 using double hashing.

Solution: when x=89

h(89)=89%10=9

Insert key 89 in hash-table in location 9

When x=18

h(18)=18%10=8

Insert key 18 in hash-table in location 8

When x=49

h(49)=49%10=9 (collision occur)

Now define new hash function as,

$h_2(x) = R - (x \bmod R)$

Or, $h_2(x) = 7 - 49\%7 = 7-0 = 7$ and

Position = (original hash value + $i*h_2(x)$) % Table size

= (49+1*7)%10 = 56%10=6

Insert key 49 in hash-table in location 6

When x=58

h(58)=58%10=8 (collision occur)

Now define new hash function as,

$h_2(x) = R - (x \bmod R)$

Or, $h_2(x) = 7 - 58\%7 = 7-2 = 5$ and

Position = (original hash value + $i*h_2(x)$) % Table size

= (58+1*5)%10 = 63%10=3

Insert key 58 in hash-table in location 3

When x=69

h(69)=69%10=9 (collision occur)

Now define new hash function as,

$h_2(x) = R - (x \bmod R)$

Or, $h_2(x) = 7 - 69\%7 = 7-6 = 1$ and

Position = (original hash value + $i*h_2(x)$) % Table size

= (69+1*1)%10 = 70%10=0

Insert key 69 in hash-table in location 0

0	1	2	3	4	5	6	7	8	9
69	None	None	58	None	None	49	None	18	89

Chaining

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases. Figure below shows the items as they are added to a hash table that uses chaining to resolve collisions.

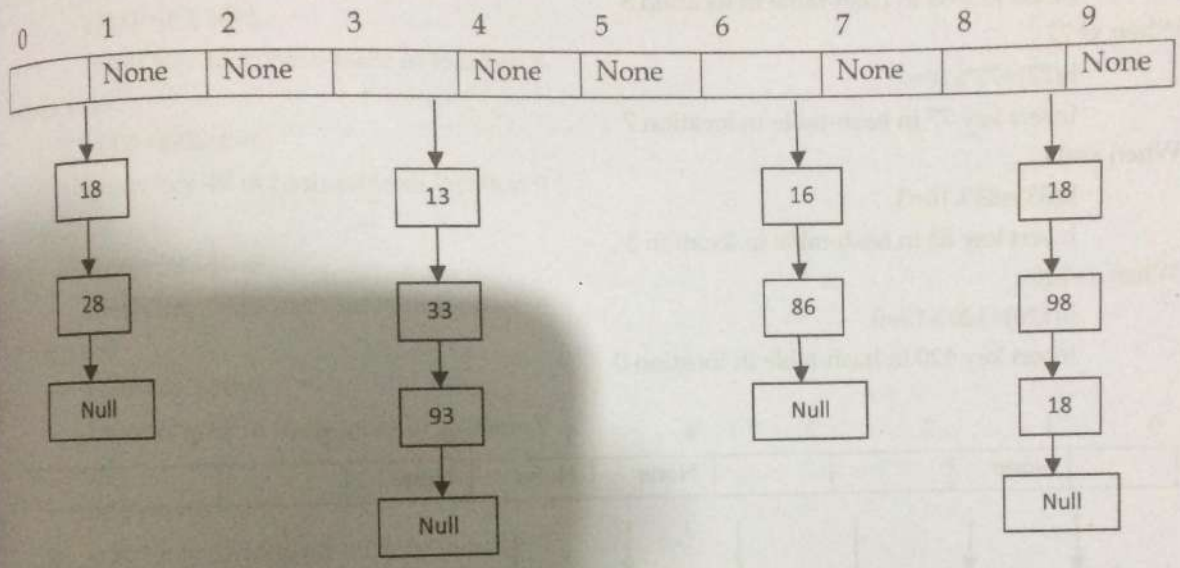


Figure: Collision Resolution with Chaining

When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient.

Example: Insert keys {102, 18, 49, 58, 69, 87, 88, 77, 83, and 120} with the hash-table size 10 using chaining method.

Solution: when x=102

h(102)=102%10=2

Insert key 102 in hash-table in location 2

When x=18

h(18)=18%10=8

Insert key 18 in hash-table in location 8

When x=49

h(49)=49%10=9

Insert key 49 in hash-table in location 9

When x=58

h(58)=58%10=8

Insert key 58 in hash-table in location 8

When x=69

h(69)=69%10=9

Insert key 69 in hash-table in location 9

When x=87

h(87)=87%10=7

Insert key 87 in hash-table in location 7

When x=88

h(88)=88%10=8

Insert key 88 in hash-table in location 8

When x=77

h(77)=77%10=7

Insert key 77 in hash-table in location 7

When x=83

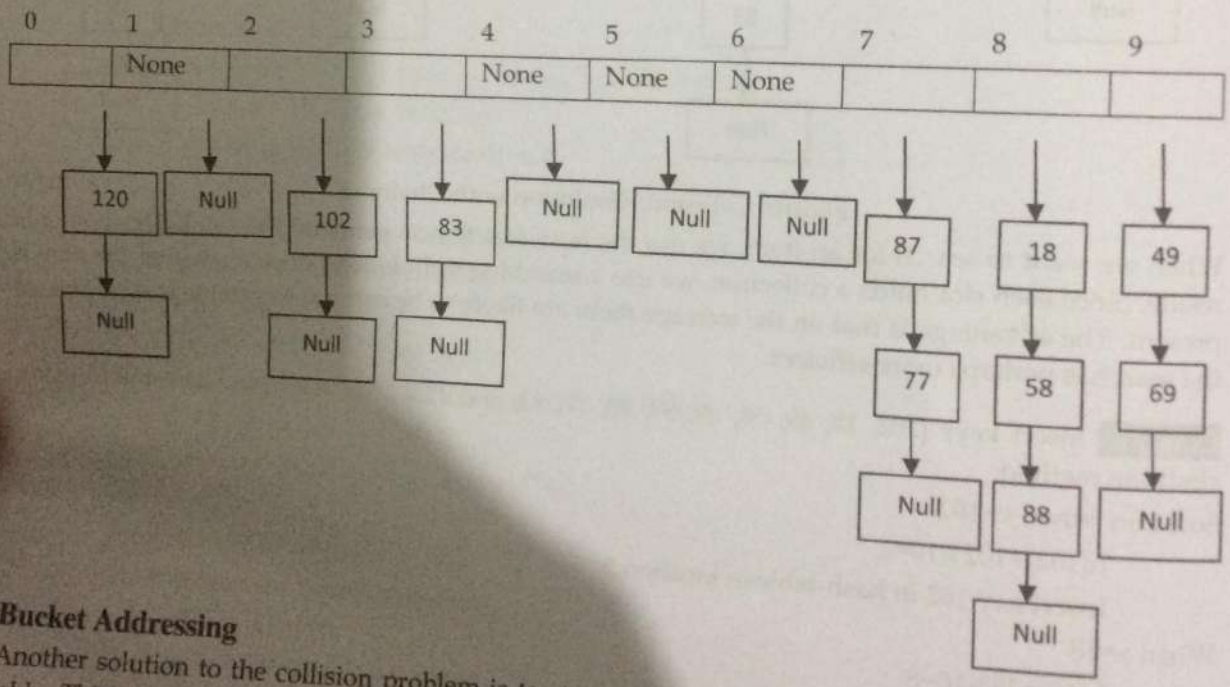
h(83)=83%10=3

Insert key 83 in hash-table in location 3

When x=120

h(120)=120%10=0

Insert key 120 in hash-table in location 0



Bucket Addressing

Another solution to the collision problem is to store colliding elements in the same position in the table. This can be achieved by associating a bucket with each address. A bucket is a block of space large enough to store multiple items. By using buckets, the problem of collisions is not totally avoided. If a bucket is already full, then an item hashed to it has to be stored somewhere else. By incorporating the open addressing approach, the colliding item can be stored in the next bucket if it has an available slot when using linear probing, or it can be stored in some other bucket when, say, quadratic probing is used.

The colliding items can also be stored in an overflow area. In this case, each bucket includes a field that indicates whether the search should be continued in this area or not. It can be simply a yes/no marker. In conjunction with chaining, this marker can be the number indicating the position in which the beginning of the linked list associated with this bucket can be found in the overflow area.

Example: Insert keys {102, 18, 49, 58, 69, 87, 88, 77, 83, 120} with the hash-table size 10 using bucket hashing.

Solution: when x=102

h(102)=102%10=2

Insert key 102 in hash-table in location 2

When x=18

h(18)=18%10=8

Insert key 18 in hash-table in location 8

When x=49

h(49)=49%10=9

Insert key 49 in hash-table in location 9

When x=58

h(58)=58%10=8

Insert key 58 in hash-table in location 8

When x=69

h(69)=69%10=9

Insert key 69 in hash-table in location 9

When x=87

h(87)=87%10=7

Insert key 87 in hash-table in location 7

When x=88

h(88)=88%10=8

Insert key 88 in hash-table in location 8

When x=77

h(77)=77%10=7

Insert key 77 in hash-table in location 7

When x=83

h(83)=83%10=3

Insert key 83 in hash-table in location 3

When x=120

h(120)=120%10=0

Insert key 120 in hash-table in location 0

0	1	2	3	4	5	6	7	8	9
120	Null	102	83	Null	Null	Null	87	18	49
Null	Null	Null	Null	Null	Null	Null	77	58	69
Null	Null	Null	Null	Null	Null	Null	Null	88	Null

Exercise

1. What do you mean by hashing? Describe hash function with suitable example.
2. What is the minimum number of keys that are hashed to their home positions using the linear probing technique?
3. Which one of the following choices gives a possible order in which the key values could have been inserted in the table?
   (A) 46, 42, 34, 52, 23, 33
   (B) 34, 42, 23, 52, 33, 46
   (C) 46, 34, 42, 23, 52, 33
   (D) 42, 46, 33, 23, 34, 52
4. Strictly speaking, the hash function used in extendible hashing also dynamically changes. In what sense is this true?
5. Apply the linear hashing method to hash numbers 12, 24, 36, 48, 60, 72, and 84 to an initially empty table with three buckets and with three cells in the overflow area. What problem can you observe? Can this problem bring the algorithm to a halt?
6. Outline an algorithm to delete a key from a table when the linear hashing method is used for inserting keys.
7. Describe double hashing with suitable example.
8. What is the drawback of using linear probing over quadratic probing? Explain.
9. What do you mean by hash collision? Explain hash collision resolving techniques with suitable example.
10. What is main drawback of using double hashing? Explain
11. Explain similarities and dissimilarities between searching techniques and hashing.
12. Here is an array with exactly 15 elements:
    [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
    Suppose that we are doing a binary search for an element. Circle any elements that will be found by examining two or fewer numbers from the array.
13. Draw a hash table with open addressing and a size of 9. Use the hash function "k%9". Insert the keys: 5, 29, 20, 0, 27 and 18 into your table (in that order).
14. Draw a hash table with chaining and a size of 9. Use the hash function "k%9" to insert the keys 5, 29, 20, 0, and 18 into your table.
15. Suppose that an open-address hash table has a capacity of 811 and it contains 81 elements. What is the table's load factor?
16. I plan to put 1000 items in a hash table, and I want the average number of accesses in a successful search to be about 2.0.
    a. About how big should the array be if I use open addressing with linear probing?
    b. About how big should the array be if I use chained hashing?
17. Here is an array with exactly 15 elements:
    [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
    Suppose that we are doing a serial search for an element. Circle any elements that will be found by examining two or fewer numbers from the array.
18. Suppose you place m items in a hash table with an array size of s. What is the correct formula for the load factor?

18. A chained hash table has an array size of 512. What is the maximum number of entries that can be placed in the table?

19. What kind of initialization needs to be done for a chained hash table? Also what is the worst-case time for binary search finding a single item in an array?

****

Program no 1: Complete menu driven program in java to perform various operations in singly linked list

```
package Linked_list;
import java.util.Scanner;
class SLLNode
{
        public int info;
        public SLLNode next;
        public SLLNode()
        {
                info=0;
                next=null;
        }
}

class SLL
{
        protected SLLNode first, last;
        public SLL()
        {
                first=null;
                last=null;
        }
        public void insertbeg(int item)
        {
                SLLNode newnode=new SLLNode();
                newnode.info=item;
                if(first==null)
                {
                        newnode.next=null;
                        first=newnode;
                        last=newnode;
                }
                else
                {
                        newnode.next=first;
                        first=newnode;
                }
        }

        public void insertEnd(int item)
        {
                SLLNode newnode=new SLLNode();
                newnode.info=item;
                if(first==null)
                {
                        first=newnode;
                        last=newnode;
                }
                else
```

```
                {
                        last.next=newnode;
                        last=newnode;
                }
        }

        public void DeleteFirst()
        {
                if(first==null)
                {
                        System.out.println("Empty linked list");
                }
                else if(first==last)
                {
                        first=null;
                        last=null;
                }
                else
                {
                        first=first.next;
                }
        }

        public void DeleteLast()
        {
                if(last==null)
                {
                        System.out.println("Empty linked list");
                }
                else if(first==last)
                {
                        first=null;
                        last=null;
                }
                else
                {
                        SLLNode temp=new SLLNode();
                        temp=first;
                        while(temp.next!=last)
                        {
                                temp=temp.next;
                        }
                        temp.next=null;
                        last=temp;
                }
        }

        public void Display()
        {
                SLLNode temp=new SLLNode();
```

```
    if(first==null)
    {
            System.out.println("Empty linked list");
    }
    else
    {
        temp=first;
        while (temp! =null)
        {
                System.out.print(temp.info+"");
                temp=temp.next;
        }
    }
  }
}

public class SLLDemo
{
        public static void main(String args[])
        {
            int choice;
            int item;
            Scanner sc=new Scanner(System.in);
            SLL sl=new SLL();
            System.out.println("1: Insert at begining");
            System.out.println("2: Insert at last");
            System.out.println("3: delete first node");
            System.out.println("4: delete last node");
            System.out.println("5: Display");
            do
            {
                    System.out.println("Enter your choice");
                    choice=sc.nextInt();
                    switch(choice)
                    {
                        case 1:
                                System.out.println("Enter data item to be inserted");
                                item=sc.nextInt();
                                sl.insertbeg(item);
                                break;
                        case 2:
                                System.out.println("Enter data item to be inserted");
                                item=sc.nextInt();
                                sl.insertEnd(item);
                                break;
                        case 3:
                                sl.DeleteFirst();
                                break;
                        case 4:
                                sl.DeleteLast();
```

```
                    break;
                case 5:
                    sl.Display();
                    break;
                default:
                System.out.println("Invalid choice Plz enter correct choice");
            }
        }while(choice<6);
    }
}
```

Output:

```
1: Insert at beginning
2: Insert at last
3: Delete first node
4: Delete last node
5: Display
Enter your choice
1
Enter data item to be inserted
43
Enter your choice
1
Enter data item to be inserted
12
Enter your choice
1
Enter data item to be inserted
88
Enter your choice
5
88 12 43 Enter your choice
2
Enter data item to be inserted
34
Enter your choice
5
88 12 43 34 Enter your choice
3
Enter your choice
5
12 43 34 Enter your choice
4
Enter your choice
5
12 43 Enter your choice
2
Enter data item to be inserted
2
Enter your choice
```

```
5
12 43 2 Enter your choice
3
Enter your choice
3
Enter your choice
3
Enter your choice
5
Empty linked list
Enter your choice
```

Program no 2: Complete menu driven Java program to implement Circular linked list

```
import java.util.Scanner;
class CLLNode
{
    public int info;
    public CLLNode next;
    public CLLNode()
    {
        info=0;
        next=null;
    }
}

class CLLDemo
{
    protected CLLNode first,last;
    public CLLDemo()
    {
        first=null;
        last=null;
    }
    public void insertbeg(int item)
    {
        CLLNode newnode=new CLLNode();
        newnode.info=item;
        if(first==null)
        {
            newnode.next=newnode;
            first=newnode;
            last=newnode;
        }
        else
        {
            newnode.next=first;
            first=newnode;
            last.next=newnode;
        }
    }
```

```
public void insertEnd(int item)
{
        CLLNode newnode=new CLLNode();
        newnode.info=item;
        if(first==null)
        {
                first=newnode;
                last=newnode;
                newnode.next=newnode;
        }
        else
        {
                last.next=newnode;
                last=newnode;
                newnode.next=first;
        }
}

public void DeleteFirst()
{
        if(first==null)
        {
                System.out.println("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
        }
        else
        {
                first=first.next;
                last.next=first;
        }
}

public void DeleteLast()
{
        if(last==null)
        {
                System.out.println("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
        }
        else
        {
```

```
                CLLNode temp=new CLLNode();
                temp=first;
                while(temp.next!=last)
                {
                        temp=temp.next;
                }
                temp.next=first;
                last=temp;
        }
    }

    public void Display()
    {
        CLLNode temp=new CLLNode();
        if(first==null)
        {
                System.out.println("Empty linked list");
        }
        else
        {
        temp=first;
        while(temp!=last)
        {
                System.out.print(temp.info+"");
                temp=temp.next;
        }
        System.out.print(last.info);
    }
}

public static void main(String args[])
{
    int choice;
    int item;
    Scanner sc=new Scanner(System.in);
    CLLDemo cl=new CLLDemo();
    System.out.println("1:Insert at begining");
    System.out.println("2:Insert at last");
    System.out.println("3:delete first node");
    System.out.println("4:delete last node");
    System.out.println("5:Display");
    do
    {
        System.out.println("Enter your choice");
        choice=sc.nextInt();
        switch(choice)
        {
            case 1:
                System.out.println("Enter data item to be inserted");
                item=sc.nextInt();
```

```
                    cl.insertbeg(item);
                    break;
            case 2:
                    System.out.println("Enter data item to be inserted");
                    item=sc.nextInt();
                    cl.insertEnd(item);
                    break;
            case 3:
                    cl.DeleteFirst();
                    break;
            case 4:
                    cl.DeleteLast();
                    break;
            case 5:
                    cl.Display();
                    break;
            default:
                    System.out.println("Invalid choice Plz enter correct choice");
            }
        }while(choice<6);
    }
}
```

Output:

```
1: Insert at beginning
2: Insert at last
3: delete first node
4: delete last node
5: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
87
Enter your choice
1
Enter data item to be inserted
32
Enter your choice
5
32 87 22 Enter your choice
2
Enter data item to be inserted
908
Enter your choice
5
32 87 22 908 Enter your choice
3
```

```
Enter your choice
5
87 22 908 Enter your choice
4
Enter your choice
5
87 22 Enter your choice
3
Enter your choice
5
22 Enter your choice
3
Enter your choice
5
Empty linked list
Enter your choice
```

Program no 3: Complete menu driven Java program to implement doubly linked list

```
import java.util.Scanner;
class DLLNode
{
        public int info;
        public DLLNode prev;
        public DLLNode next;
        public DLLNode()
        {
                info=0;
                prev=null;
                next=null;
        }
}

class DLLDemo
{
        protected DLLNode first,last;
        public DLLDemo()
        {
                first=null;
                last=null;
        }
        public void insertbeg(int el)
        {
                DLLNode newnode=new DLLNode();
                newnode.info=el;
                newnode.prev=newnode.next=null;
                if(first==null)
                {
                        first=newnode;
                        last=newnode;
                }
```

```
            else
            {
                    newnode.next=first;
                    first.prev=newnode;
                    first=newnode;
            }
    }

    public void insertEnd(int el)
    {
            DLLNode newnode=new DLLNode();
            newnode.info=el;
            newnode.prev=newnode.next=null;
            if(first==null)
            {
                    first=newnode;
                    last=newnode;
            }
            else
            {
                    last.next=newnode;
                    newnode.prev=last;
                    last=newnode;
            }
    }

    public void DeleteFirst()
    {
            if(first==null)
            {
                    System.out.println("Empty linked list");
            }
            else if(first==last)
            {
                    first=null;
                    last=null;
            }
            else
            {
                    first=first.next;
            }
    }

    public void DeleteLast()
    {
            if(last==null)
            {
                    System.out.println("Empty linked list");
            }
            else if(first==last)
```

```
        {
                first=null;
                last=null;
        }
        else
        {
                DLLNode temp=new DLLNode();
                temp=first;
                while(temp.next!=last)
                {
                        temp=temp.next;
                }
                temp.next=null;
                last=temp;
        }
}

public void Display()
{
        DLLNode temp=new DLLNode();
        if(first==null)
        {
                System.out.println("Empty linked list");
        }
        else
        {
                temp=first;
                while(temp!=last)
                {
                        System.out.print(temp.info+"");
                        temp=temp.next;
                }
                System.out.print(last.info);
        }
}

public static void main(String args[])
{
        int choice;
        int item;
        Scanner sc=new Scanner(System.in);
        DLLDemo dl=new DLLDemo();
        System.out.println("1:Insert at begining");
        System.out.println("2:Insert at last");
        System.out.println("3:delete first node");
        System.out.println("4:delete last node");
        System.out.println("5:Display");
        do
        {
                System.out.println("Enter your choice");
```

```
                    choice=sc.nextInt();
                    switch(choice)
                    {
                        case 1:
                            System.out.println("Enter data item to be inserted");
                            item=sc.nextInt();
                            dl.insertbeg(item);
                            break;
                        case 2:
                            System.out.println("Enter data item to be inserted");
                            item=sc.nextInt();
                            dl.insertEnd(item);
                            break;
                        case 3:
                            dl.DeleteFirst();
                            break;
                        case 4:
                            dl.DeleteLast();
                            break;

                        case 5:
                            dl.Display();
                            break;
                        default:
                        System.out.println("Invalid choice Plz enter correct choice");
                    }
                }while(choice<6);
            }
        }
```

Output:

```
1: Insert at beginning
2: Insert at last
3: delete first node
4: delete last node
5: Display
Enter your choice
1
Enter data item to be inserted
33
Enter your choice
2
Enter data item to be inserted
77
Enter your choice
1
Enter data item to be inserted
98
Enter your choice
5
```

```
98 33 77 Enter your choice
2
Enter data item to be inserted
54
Enter your choice
5
98 33 77 54 Enter your choice
2
Enter data item to be inserted
5
Enter your choice
5
98 33 77 54 5 Enter your choice
3
Enter your choice
5
33 77 54 5 Enter your choice
4
Enter your choice
5
33 77 54 Enter your choice
4
Enter your choice
5
33 77 Enter your choice
3
Enter your choice
5
77 Enter your choice
3
Enter your choice
5
Empty linked list
Enter your choice
```

Program no 4: Complete menu driven Java program to implement circular doubly linked list

```
import java.util.Scanner;
class CDLLNode
{
        public int info;
        public CDLLNode prev;
        public CDLLNode next;
        public CDLLNode()
        {
                info=0;
                prev=null;
                next=null;
        }
```

```
class CDLLDemo
{
        protected CDLLNode first,last;
        public CDLLDemo()
        {
                first=null;
                last=null;
        }

        public void insertbeg(int el)
        {
                CDLLNode newnode=new CDLLNode();
                newnode.info=el;
                if(first==null)
                {
                        first=newnode;
                        last=newnode;
                        newnode.next=newnode;
                        newnode.prev=newnode;
                }
                else
                {
                        newnode.next=first;
                        first.prev=newnode;
                        first=newnode;
                        last.next=first;
                        first.prev=last;
                }
        }

        public void insertEnd(int el)
        {
                CDLLNode newnode=new CDLLNode();
                newnode.info=el;
                if(first==null)
                {
                        first=newnode;
                        last=newnode;
                        newnode.next=newnode;
                        newnode.prev=newnode;
                }
                else
                {
                        last.next=newnode;
                        newnode.prev=last;
                        last=newnode;
                        last.next=first;
                        first.prev=last;
                }
        }
```

```
public void DeleteFirst()
{
        if(first==null)
        {
                System.out.println("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
        }
        else
        {
                first=first.next;
                last.next=first;
                first.prev=last;
        }
}

public void DeleteLast()
{
        if(last==null)
        {
                System.out.println("Empty linked list");
        }
        else if(first==last)
        {
                first=null;
                last=null;
        }
        else
        {
                CDLLNode temp=new CDLLNode();
                temp=first;
                while(temp.next!=last)
                {
                        temp=temp.next;
                }
                last=temp;
                last.next=first;
                first.prev=last;
        }
}
public void Display()
{
        CDLLNode temp=new CDLLNode();
        if(first==null)
        {
                System.out.println("Empty linked list");
```

```
        }
        else
        {
                temp=first;
                while(temp!=last)
                {
                        System.out.print(temp.info+"");
                        temp=temp.next;
                }
                System.out.print(last.info);
        }
}

public static void main(String args[])
{
        int choice;
        int item;
        Scanner sc=new Scanner(System.in);
        CDLLDemo cdl=new CDLLDemo();
        System.out.println("1:Insert at begining");
        System.out.println("2:Insert at last");
        System.out.println("3:delete first node");
        System.out.println("4:delete last node");
        System.out.println("5:Display");
        do
        {
                System.out.println("Enter your choice");
                choice=sc.nextInt();
                switch(choice)
                {
                        case 1:
                                System.out.println("Enter data item to be inserted");
                                item=sc.nextInt();
                                cdl.insertbeg(item);
                                break;
                        case 2:
                                System.out.println("Enter data item to be inserted");
                                item=sc.nextInt();
                                cdl.insertEnd(item);
                                break;
                        case 3:
                                cdl.DeleteFirst();
                                break;
                        case 4:
                                cdl.DeleteLast();
                                break;
                        case 5:
                                cdl.Display();
                                break;
                        default:
```

```
                    System.out.println("Invalid choice Plz enter correct choice");
                    }
                }while(choice<6);
            }
        }
```

Output:

```
1: Insert at beginning
2: Insert at last
3: delete first node
4: delete last node
5: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
99
Enter your choice
5
99 22 Enter your choice
2
Enter data item to be inserted
16
Enter your choice
5
99 22 16 Enter your choice
3
Enter your choice
5
22 16 Enter your choice
4
Enter your choice
5
22 Enter your choice
3
Enter your choice
5
Empty linked list
Enter your choice
```

Program no 5: Complete java program to implement the skip list

```
import java.util.Random;
public class IntSkipListNode
{
        public int key;
        public IntSkipListNode[ ] next;
        IntSkipListNode(int i, int n)
        {
```

```
            key = i;
            next = new IntSkipListNode[n];
            for (int j = 0; j < n; j++)
                next[j] = null;
        }
}

public class IntSkipList
{
        private int maxLevel;
        private IntSkipListNode[ ] root;
        private int[ ] powers;
        private Random rd = new Random();
        IntSkipList()
        {
            this(4);
        }
        IntSkipList (int i)
        {
            maxLevel = i;
            root = new IntSkipListNode[maxLevel];
            powers = new int[maxLevel];
            for (int j = 0; j < maxLevel; j++)
                root[j] = null;
            choosePowers();
        }
        public boolean isEmpty()
        {
            return root[0] == null;
        }
        public void choosePowers()
        {
            powers[maxLevel-1] = (2 << (maxLevel-1)) - 1;          // 2^maxLevel - 1
            for (int i = maxLevel - 2, j = 0; i >= 0; i--, j++)
                powers[i] = powers[i+1] - (2 << j); // 2^(j+1)
        }
        public int chooseLevel()
        {
            int i, r = Math.abs(rd.nextInt()) % powers[maxLevel-1] + 1;
            for (i = 1; i < maxLevel; i++)
                if (r < powers[i])
                    return i-1; // return a level < the highest level;
            return i-1; // return the highest level;
        }
        public int skipListSearch (int key)
        {
            int lvl;
            IntSkipListNode prev, curr; // find the highest non-null
            for (lvl = maxLevel-1; lvl >= 0 && root[lvl] == null; lvl--); // level;
            prev = curr = root[lvl];
```

```
        while (true)
        {
                if (key == curr.key) // success if equal;
                        return curr.key;
                else if (key < curr.key)   // if smaller, go down,
                {
                        if (lvl == 0) // if possible,
                                return 0;
                        else if (curr == root[lvl]) // by one level
                                curr = root[--lvl]; // starting from the
                        else curr = prev.next[--lvl]; // predecessor which
                } // can be the root;
                else
                {
                        prev = curr; // go to the next
                        if (curr.next[lvl] != null) // non-null node
                                curr = curr.next[lvl]; // on the same level
                        else
                        {
                                for (lvl--; lvl >= 0 && curr.next[lvl] == null; lvl--);
                                        if (lvl >= 0)
                                                curr = curr.next[lvl];
                                        else
                                                return 0;
                        }
                }
        }
}

public void skipListInsert (int key)
{
        IntSkipListNode[ ] curr = new IntSkipListNode[maxLevel];
        IntSkipListNode[ ] prev = new IntSkipListNode[maxLevel];
        IntSkipListNode newNode;
        int lvl, i;
        curr[maxLevel-1] = root[maxLevel-1];
        prev[maxLevel-1] = null;
        for (lvl = maxLevel - 1; lvl >= 0; lvl--)
        {
                while (curr[lvl] != null && curr[lvl].key < key)
                {
                        prev[lvl] = curr[lvl]; // if smaller;
                        curr[lvl] = curr[lvl].next[lvl];
                }
                if (curr[lvl] != null && curr[lvl].key == key)
                        return;
                if (lvl > 0) // go one level down
                        if (prev[lvl] == null)   // if not the lowest
                        {
                                curr[lvl-1] = root[lvl-1]; // level, using a link
```

```
                                        prev[lvl-1] = null; // either from the root
                                }
                                else
                                {
                                        curr[lvl-1] = prev[lvl].next[lvl-1];
                                        prev[lvl-1] = prev[lvl];
                                }
                }
                lvl = chooseLevel(); // generate randomly level
                newNode = new IntSkipListNode(key,lvl+1);
                for (i = 0; i <= lvl; i++)
                {
                        newNode.next[i] = curr[i]; // newNode and reset to newNode
                        if (prev[i] == null) // either fields of the root
                                root[i] = newNode; // or next fields of newNode's
                        else
                                prev[i].next[i] = newNode;// predecessors;
                }
        }
}
```

Program no 6: Complete menu driven Java program to array implementation of stack

```
import java.util.Scanner;
class Stack_structure
{
        final int SIZE=10;
        public intitem[]=new int[SIZE];
        public int top;
        public Stack_structure()
        {
                top=-1;
        }
        booleanIsFull()
        {
                return top==SIZE-1;
        }
        booleanIsEmpty()
        {
          return top==-1;
        }
}
public class StackDemo
{
        Stack_structure s=new Stack_structure();
        public void push(int el)
        {
                if(!s.IsFull())
                {
                        s.top++;
                        s.item[s.top]=el;
```

```
        }
        else
        {
                System.out.println("Stack full");
        }
}
public void pop()
{
        if(s.IsEmpty())
        {
                System.out.println("Stack Empty");
        }
        else
        {
                int el;
        el=s.item[s.top];
                s.top--;
                System.out.println("Popped element="+el);
        }
}
public void display()
{
        if(s.IsEmpty())
        {
                System.out.println("Stack empty");
        }
        else
        {
                int i;
                System.out.println("Stack elements are:");
        for(i=0;i<=s.top;i++)
        {
                        System.out.print(s.item[i]+" ");
        }
        }
}
public static void main(String args[])
{
        int choice;
        int el;
        Scanner sc=new Scanner(System.in);
        StackDemost=new StackDemo();
        System.out.println("1:Push");
        System.out.println("2:Pop");
        System.out.println("3:Display");
        do
        {
                System.out.println("Enter your choice");
        choice=sc.nextInt();
        switch(choice)
```

```
{
        case 1:
                System.out.println("Enter data item to be inserted");
                el=sc.nextInt();
                st.push(el);
                break;
        case 2:
                st.pop();
                break;
        case 3:
                st.display();
                break;
        default:
        System.out.println("Invalid choice Please enter correct choice");
}
}while(choice<4);
}
}
```

Output:

```
1: Push
2: Pop
3: Display
Enter your choice
1
Enter data item to be inserted
99
Enter your choice
1
Enter data item to be inserted
89
Enter your choice
1
Enter data item to be inserted
88
Enter your choice
1
Enter data item to be inserted
90
Enter your choice
3
Stack elements are:
99 89 88 90 Enter your choice
2
Popped element=90
Enter your choice
2
Popped element=88
Enter your choice
2
Popped element=89
```

```
Enter your choice
1
Enter data item to be inserted
45
Enter your choice
3
Stack elements are:
99 45 Enter your choice
2
Popped element=45
Enter your choice
2
Popped element=99
Enter your choice
2
Stack Empty
Enter your choice
```

Program no 7: Complete java program tom implement delimiter matching program by using stack data structure:

```
import java.io.*;
class StackX
{
    private int maxSize;
    private char[ ] stackArray;
    private int top;
    public StackX(int s) // constructor
    {
        maxSize = s;
        stackArray = new char[maxSize];
        top = -1;
    }
    public void push(char j) // put item on top of stack
    {
        stackArray[++top] = j;
    }

    public char pop() // take item from top of stack
    {
        return stackArray[top--];
    }

    public char peek() // peek at top of stack
    {
        return stackArray[top];
    }

    public boolean isEmpty() // true if stack is empty
    {
        return (top == -1);
```

```
        }
} // end class StackX

class BracketChecker
{
        private String input; // input string
        public BracketChecker(String in) // constructor
        {
                input = in;
        }
        public void check()
        {
                int stackSize = input.length(); // get max stack size
                StackX theStack = new StackX(stackSize);
                for(int j=0; j<input.length(); j++)
                {
                        char ch = input.charAt(j);
                        switch(ch)
                        {
                                case '{':
                                case '[':
                                case '(':
                                        theStack.push(ch);
                                        break;
                                case '}': // closing symbols
                                case ']':
                                case ')':
                                if( !theStack.isEmpty() ) // if stack not empty,
                                {
                                        char chx = theStack.pop();
                                        if( (ch=='}' && chx!='{') || (ch==']' && chx!='[') ||
                                        (ch==')' && chx!='(') )
                                                System.out.println("Error: "+ch+" at "+j);
                                }
                                else // prematurely empty
                                        System.out.println("Error: "+ch+" at "+j);
                                        break;
                                default: // no action on other characters
                                        break;
                        }
                }
                if( !theStack.isEmpty() )
                        System.out.println("Error: missing right delimiter");
        }
} // end class BracketChecker

class BracketsApp
{
        public static void main(String[ ] args) throws IOException
        {
```

```
            String input;
            while(true)
            {
                    System.out.print("Enter string containing delimiters: ");
                    System.out.flush();
                    input = getString();
                    if( input.equals(" ") ) // quit if [Enter]
                            break;
                    BracketChecker theChecker = new BracketChecker(input);
                    theChecker.check();
            }
    }
    public static String getString() throws IOException
    {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            String s = br.readLine();
            return s;
    }
}// end class BracketsApp
```

Program no 8: Linked list implementation of stack

```
import java.util.Scanner;
class Node
{
        public int info;
        public Node next;
        public Node()
        {
                info=0;
                next=null;
        }
}
class Linked_stack
{
        protected Node top=new Node();
        public Linked_stack ()
        {
                top=null;
        }
        public void push(int item)
        {
                Node newnode=new Node();
                newnode.info=item;
                if(top==null)
                {
                        newnode.next=null;
                top=newnode;
                }
                else
```

```
            {
                newnode.next=top;
                top=newnode;
            }
        }
        /*pop function*/
        public void pop()
        {
            if(top==null)
            {
                System.out.println("Stack empty");
            }
            else
            {
            Node temp=new Node();
            temp=top;
            top=top.next;
                System.out.println("Poped item="+temp.info);
            }
        }

        public void Display()
        {
            if(top==null)
            {
                System.out.println("Empty stack");
            }
            else
            {
            Node temp=new Node();
            temp=top;
            while(temp!=null)
            {
                System.out.print(temp.info+" ");
                temp=temp.next;
            }
        }
    }

    public static void main(String args[])
    {
        int choice;
        int el;
        Scanner sc=new Scanner(System.in);
        Linked_stackst=new Linked_stack();
        System.out.println("1:Push");
        System.out.println("2:Pop");
        System.out.println("3:Display");
        do
        {
```

```
            System.out.println("Enter your choice");
        choice=sc.nextInt();
        switch(choice)
        {
                case 1:
                        System.out.println("Enter data item to be inserted");
                        el=sc.nextInt();
                        st.push(el);
                        break;
                case 2:
                        st.pop();
                        break;
                case 3:
                        st.Display();
                        break;
                default:
                        System.out.println("Invalid choice Plz enter correct choice");
        }
        }while(choice<4);
    }
}
```

Output:

```
1: Push
2: Pop
3: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
44
Enter your choice
1
Enter data item to be inserted
55
Enter your choice
3
55 44 22 Enter your choice
2
Popped item=55
Enter your choice
2
Popped item=44
Enter your choice
2
Popped item=22
Enter your choice
2
```

```
Stack empty
Enter your choice
1
Enter data item to be inserted
33
Enter your choice
3
33 Enter your choice.....
```

Program no 9: Complete Java program to convert an expression from infix to postfix:

```
package Linked_list;
import java.util.Scanner;
class Operator_precedency
{
intprecedency(char ch)
  {
    switch(ch)
    {
      case '$':
                return(4);
      case'*':
      case'/':
                return(3);
      case'+':
      case'-':
                return(2);
      default:
                return(1);
    }
  }
}
class infixtopostfix
{
  final int SIZE=100;
  public static void main(String args[])
  {

      String infix=new String();
      char poststack[]=new char[100];
      char opstack[]=new char[100];
      intoptop=-1,ptop=-1;
      Operator_precedency p=new Operator_precedency();
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter valid infix expression");
      infix=sc.nextLine();
      int length=(infix.length());
      intlen=length;
      for(inti=0;i<length;i++)
      {
if(infix.charAt(i)=='(')
{
```

```
            opstack[++optop]=infix.charAt(i);
            len--;
    }
    elseif(infix.charAt(i)=='+'||infix.charAt(i)=='-'||infix.charAt(i)=='*'||infix.charAt(i)
    ==='/'||infix.charAt(i)=='$')
    {
      if(optop!=-1)
      {
        while(p.precedency(opstack[optop])>p.precedency(infix.charAt(i)))
        {
                    poststack[++ptop]=opstack[optop--];
        }
            opstack[++optop]=infix.charAt(i);
      }
      else
       {
                    opstack[++optop]=infix.charAt(i);
      }
    }
    else if(infix.charAt(i)==')')
    {
            len--;
            while(opstack[optop]!='(')
            {
                    poststack[++ptop]=opstack[optop];
                    optop--;
            }
            optop--;
    }
    else
    {
      poststack[++ptop]=infix.charAt(i);
    }
  }
  while(optop!=-1)
  {
      poststack[++ptop]=opstack[optop];
      optop--;
  }
      for(inti=0; i<=len; i++)
      {
            System.out.print(poststack[i]);
  }
 }
}
```

Output:

```
Enter valid infix expression
(A+B*C/D)*(E-F/G)
ABCD/*+EFG/-*
```

Program no 10: Complete Java program for evaluating postfix expression

```
package Linked_list;
import java.util.Scanner;
class PostDemo
{
        PostfixEvaluation p=new PostfixEvaluation();
        int pop( )
        {
                int n;
                n=p.vstack[p.top--];
                return(n);
        }
        void push(intval)
        {
                p.vstack[++p.top]=val;
        }
}
public class PostfixEvaluation
{
        intvstack[]=new int[100];
        int top=-1;
        public static void main(String args[])
        {
                String postfix=new String();
                char ch;
                int i, res, len, op1, op2;
                int value[]=new int[100];
                PostDemo p=new PostDemo();
                Scanner sc=new Scanner(System.in);
                System.out.println("Enter valid postfix expression");
                postfix=sc.nextLine();
                len=postfix.length();
                for(i=0;i<len;i++)
                {
                        if(postfix.charAt(i)=='+'||postfix.charAt(i)=='-'||postfix.charAt(i)
                        =='*'|| postfix.charAt(i)=='/'|| postfix.charAt(i)=='$')
                        {
                                ch=postfix.charAt(i);
                                op2=p.pop();
                                op1=p.pop();
                                switch(ch)
                                {
                                        case '+':
                                                p.push(op1+op2);
                                                break;
                                        case'-':
                                                p.push(op1-op2);
                                                break;
                                        case'*':
                                                p.push(op1*op2);
```

```
                    break;
                case'/':
                    p.push(op1/op2);
                    break;
                case'$':
                    p.push((int) Math.pow(op1,op2));
                    break;
                case'%':
                    p.push(op1%op2);
                    break;
                }
            }
            else //if operands
            {
                System.out.println("Enter value of "+postfix.charAt(i));
                value[i]=sc.nextInt();
                p.push(value[i]);
            }
        }
        System.out.print("The result is:");
        res=p.pop();
        System.out.println(res);
    }
}
```

Output:

```
Enter valid postfix expression
ab*c-
Enter value of a
10
Enter value of b
2
Enter value of c
5
The result is: 15
```

Program no 11: Calculation of the factorial of an integer number using recursive function

```
import java.util.Scanner;
class FactDemo
{
  public intfactorial(int n)
  {
    if(n==0)
      return 1;
    else
      return (n*factorial(n-1));
  }
}
public class Factorial
{
  public static void main(String args[])
```

```
    {
        int n;
        int fact;
        Scanner sc=new Scanner(System.in);
        FactDemo f=new FactDemo();
        System.out.println("Enter value of n:");
        n=sc.nextInt();
        fact=f.factorial(n);
        System.out.println("Factorial of "+n+"="+fact);
    }
}
```

Output:

Enter value of n:
6
Factorial of 6=720

Program no 12: Program to generate Fibonacci series up to n terms using recursive function. (Hint Fibonacci sequence=0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,.....)

```
import java.util.Scanner;
class Fibonacci
{
    public intfibo(int n)
    {
        if(n== 1 || n == 2)
        return 1;
        else
        return fibo(n-1)+fibo(n-2);
    }
}

public class FibonacciSeries
{
    public static void main(String args[])
    {
        int n, i;
        Scanner sc=new Scanner(System.in);
        Fibonacci f=new Fibonacci();
        System.out.println("Enter value of n:");
        n=sc.nextInt();
        System.out.println("Fibonacci numbers up"+n+"th term is:");
        for(i=1; i<=n; i++)
            System.out.print(f.fibo(i)+" ");
    }
}
```

Output:

Enter value of n:
7
Fibonacci numbers up7th term is:
1 1 2 3 5 8 13

Program no 13: Program to find nth tern of Fibonacci series using recursion

```
import java.util.Scanner;
class Fibonacci
{
  public intfibo(int n)
  {
        if(n== 1 || n == 2)
        return 1;
        else
        return fibo(n-1)+fibo(n-2);
  }
}

public class FibonacciSeries
{
  public static void main(String args[])
  {
        int n;
        Scanner sc=new Scanner(System.in);
        Fibonacci f=new Fibonacci();
        System.out.println("Enter value of n:");
        n=sc.nextInt();
        System.out.print(n+"th term of Fibonacci series is:"+f.fibo(n));
  }
}
```

Output:

```
Enter value of n:
10
10th term of Fibonacci series is: 55
```

Program no 14: Program to find sum of first n natural numbers using recursion

```
import java.util.Scanner;
class Natura_sum_Demo
{
  public int sum(int n)
  {
    if(n==1)
      return 1;
    else
      return (n+sum(n-1));
  }
}
public class Natural_sum
{
  public static void main(String args[])
  {
        int n;
        int s;
        Scanner sc=new Scanner(System.in);
        Natura_sum_Demo ns=new Natura_sum_Demo ();
        System.out.println("Enter value of n:");
```

```
        n=sc.nextInt();
        s=ns.sum(n);
        System.out.println("Sum of "+n+"+"natural sum"+="+s);
    }
}
```

Program no 15: Recursive java program to display following pattern

```
*
*     *
*     *     *
*     *     *     *
*     *     *     *     *
*     *     *     *     *     *
*     *     *     *     *     *     *
```

```
import java.util.Scanner;
class Pattern_Demo
{
    public int display(int n)
    {
        if (n <= 0)
            return;
        else
            display(n - 1);
        for (int i = 1; i <= n; i++)
        {
            System.out.print("*"+" ");
        }
        System.out.println(" ");
    }
}
public class Pattern
{
    public static void main(String args[])
    {
        Pattern_Demo p=new Pattern_Demo();
        p.display(7);
    }
}
```

Program no 16: Recursive solution of tower of Hanoi

```
import java.util.Scanner;
class TOHSimulation
{
    void TOH(int n, char A, char B, char C)
    {
        if(n>0)
        {
            TOH(n-1, A, C, B);
            System.out.println("Move disk "+n+" from "+A+" to "+B);
            TOH(n-1, C, B, A);
```

```
        }
    }
}
public class TOHDemo
{
    public static void main(String args[])
    {
        int n;
        Scanner sc=new Scanner(System.in);
        TOHSimulation t=new TOHSimulation();
        System.out.println("Enter number of disks");
        n=sc.nextInt();
        t.TOH(n,'O','D','I');
    }
}
```

Output:

```
Enter number of disks
3
Move disk 1 from O to D
Move disk 2 from O to I
Move disk 1 from D to I
Move disk 3 from O to D
Move disk 1 from I to O
Move disk 2 from I to D
Move disk 1 from O to D
```

Program no 17: Complete java program to showing the use of tail recursion

```
import java.util.Scanner;
class fact1
{
        fact2 f2=new fact2();
        public int factorial(int n)
        {
                return f2.fact(n,1);
        }
}

class fact2
{
        public int fact (int n, int accumulator)
        {
                if (n == 0)
                        return accumulator;
                else
                        return fact(n - 1, n * accumulator);
        }
}
public class TailRecursion
{
        public static void main(String args[])
        {
```

```
            int num, f;
            Scanner sc=new Scanner(System.in);
            fact1 f1=new fact1();
            System.out.println("Enter any number");
            num=sc.nextInt();
            f=f1.factorial(num);
            System.out.println("Factorial of given number= "+f);
        }
}
```

Output

```
Enter any number
10
Factorial of given number= 3628800
```

Program no 18: Menu driven program for array implementation of linear queue in Java

```
import java.util.Scanner;
class Queue_Structure
{
   final int SIZE=10;
   public int item[]=new int[SIZE];
   public int rear,front;
   public Queue_Structure()
   {
      rear=-1;
      front=0;
   }
   boolean IsFull()
   {
      return rear==SIZE-1;
   }
   boolean IsEmpty()
   {
      return rear<front;
   }
}

public class QueueDemo
{
   Queue_Structure q=new Queue_Structure();
   public void Enqueue(int el)
   {
      if(!q.IsFull())
      {
         q.rear++;
         q.item[q.rear]=el;
      }
      else
      {
         System.out.println("Queue full");
      }
}
```

```
public void Dequeue()
{
  if(q.IsEmpty())
  {
    System.out.println("Queue Empty");
  }
  else
  {
    int el;
    el=q.item[q.front];
    q.front++;
    System.out.println("Deleted element="+el);
  }
}

public void display()
{
  if(q.IsEmpty())
  {
    System.out.println("Queue empty");
  }
  else
  {
    int i;
    System.out.println("Queue elements are:");
    for(i=q.front;i<=q.rear;i++)
    {
      System.out.print(q.item[i]+" ");
    }
  }
}
public static void main(String args[])
{
  int choice;
  int el;
  Scanner sc=new Scanner(System.in);
  QueueDemo st=new QueueDemo();
  System.out.println("1: Enqueue");
  System.out.println("2: Dequeue");
  System.out.println("3: Display");
  do
  {
    System.out.println("Enter your choice");
    choice=sc.nextInt();
    switch(choice)
    {
      case 1:
        System.out.println("Enter data item to be inserted");
        el=sc.nextInt();
        st.Enqueue(el);
```

```
            break;
          case 2:
            st.Dequeue();
            break;
          case 3:
            st.display();
            break;
          default:
            System.out.println("Invalid choice Plz enter correct choice");
        }
      }while(choice<4);
    }
  }
```

Output:

```
1: Enqueue
2: Dequeue
3: Display
Enter your choice
1
Enter data item to be inserted
99
Enter your choice
1
Enter data item to be inserted
55
Enter your choice
1
Enter data item to be inserted
45
Enter your choice
3
Queue elements are:
99 55 45 Enter your choice
2
Deleted element=99
Enter your choice
2
Deleted element=55
Enter your choice
2
Deleted element=45
Enter your choice
3
Queue empty
Enter your choice
1
Enter data item to be inserted
24
Enter your choice
1
```

```
Enter data item to be inserted
56
Enter your choice
3
Queue elements are:
24 56 Enter your choice
2
Deleted element=24
Enter your choice
```

Program no 19: Linked list implementation of linear queue

```
package Linked_list;
import java.util.Scanner;
class QNode
{
  public int info;
  public QNode next;
  public QNode()
  {
    info=0;
    next=null;
  }
}

class Linked_LinearQueue
{
  QNode rear=new QNode();
  QNode front=new QNode();
  public Linked_LinearQueue()
  {
    rear=null;
    front=null;
  }

  public void Enqueue(int item)
  {
    QNode newnode=new QNode();
    newnode.info=item;
    newnode.next=null;
    if(rear==null)
    {
      rear=front=newnode;
    }
    else{
      rear.next=newnode;
      rear=newnode;
    }
  }

  public void Dequeue()
```

```
{
  if(front==null)
  {
    System.out.println("Empty queue");
  }
  else if(rear==front)
  {
    System.out.println("Deleted item="+front.info);
    rear=front=null;
  }
  else
  {
    QNode temp=new QNode();
    temp=front;
    front=front.next;
    System.out.println("Deleted item="+temp.info);
  }
}

public void Display()
{
  if(front==null)
  {
    System.out.println("Empty queue");
  }
  else
  {
    QNode temp=new QNode();
    temp=front;
    while(temp!=null)
    {
      System.out.print(temp.info+" ");
      temp=temp.next;
    }
  }
}

public static void main(String args[])
{
  int choice;
  int el;
  Scanner sc=new Scanner(System.in);
  Linked_LinearQueue st=new Linked_LinearQueue ();
  System.out.println("1:Enqueue");
  System.out.println("2:Dequeue");
  System.out.println("3:Display");
  do
  {
  System.out.println("Enter your choice");
  choice=sc.nextInt();
```

```
      switch(choice)
      {
        case 1:
          System.out.println("Enter data item to be inserted");
          el=sc.nextInt();
          st.Enqueue(el);
          break;
        case 2:
          st.Dequeue();
          break;
        case 3:
          st.Display();
          break;
        default:
          System.out.println("Invalid choice Plz enter correct choice");
      }
    }while(choice<4);
  }
}
```

Program no 20: Array implementation of circular queue with scarifying one cell

```
import java.util.Scanner;
class CQueue_Structure
{
  final int SIZE=5;
  public int item[]=new int[SIZE];
  public int rear,front;
  public CQueue_Structure()
  {
    rear=SIZE-1;
    front=SIZE-1;
  }
  boolean IsFull()
  {
    return (front==(rear+1)%SIZE);
  }
  boolean IsEmpty()
  {
    return rear==front;
  }
}

public class Circular_QueueDemo
{
  CQueue_Structure q=new CQueue_Structure();
  public void Enqueue(int el)
  {
    if(q.IsFull())
    {
      System.out.println("Queue full");
```

```
    }
    else
    {
      q.rear=(q.rear+1)%q.SIZE;
      q.item[q.rear]=el;
    }
  }

  public void Dequeue()
  {
    if(q.IsEmpty())
    {
      System.out.println("Queue Empty");
    }
    else
    {
      int el;
      q.front=(q.front+1)%q.SIZE;
      el=q.item[q.front];
      System.out.println("Deleted element="+el);
    }
  }
  public void display()
  {
    if(q.IsEmpty())
    {
      System.out.println("Queue empty");
    }
    else
    {
      int i;
      System.out.println("Queue elements are:");
      for(i=(q.front+1)%q.SIZE;i!=q.rear;i=(i+1)%q.SIZE)
      {
        System.out.print(q.item[i]+" ");
      }
      System.out.print(q.item[q.rear]+" ");
    }
  }
  public static void main(String args[])
  {
    int choice;
    int el;
    Scanner sc=new Scanner(System.in);
    Circular_QueueDemo st=new Circular_QueueDemo();
    System.out.println("1:Enqueue");
    System.out.println("2:Dequeue");
    System.out.println("3:Display");
    do
    {
```

```
        System.out.println("Enter your choice");
        choice=sc.nextInt();
        switch(choice)
        {
          case 1:
            System.out.println("Enter data item to be inserted");
            el=sc.nextInt();
            st.Enqueue(el);
            break;
          case 2:
            st.Dequeue();
            break;
          case 3:
            st.display();
            break;
          default:
            System.out.println("Invalid choice Plz enter correct choice");
        }
      }while(choice<4);
    }
}
```

Output:

```
1: Enqueue
2: Dequeue
3: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
54
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
77
Enter your choice
1
Enter data item to be inserted
99
Queue full
Enter your choice
3
Queue elements are:
22 54 22 77 Enter your choice
```

```
2
Deleted element=22
Enter your choice
2
Deleted element=54
Enter your choice
2
Deleted element=22
Enter your choice
1
Enter data item to be inserted
43
Enter your choice
3
Queue elements are:
77 43 Enter your choice
```

Program no 21: Array Implementation of circular queue without sacrificing one cell by using a count variable

```
package Linked_list;
import java.util.Scanner;
class CQueue_Struc
{
  final int SIZE=5;
  public int item[]=new int[SIZE];
  public int rear,front;
  int count=0;
  public CQueue_Struc()
  {
    rear=SIZE-1;
    front=SIZE-1;
  }
  boolean IsFull()
  {
    return count==SIZE;
  }
  boolean IsEmpty()
  {
    return count==0;
  }
}
public class CQ_WithCount
{
  CQueue_Struc q=new CQueue_Struc();
  public void Enqueue(int el)
  {
    if(q.IsFull())
    {
      System.out.println("Queue full");
    }
```

```
else
{
  q.count++;
  q.rear=(q.rear+1)%q.SIZE;
  q.item[q.rear]=el;
}
}
public void Dequeue()
{
  if(q.IsEmpty())
  {
    System.out.println("Queue Empty");
  }
  else
  {
    int el;
    q.count--;
    q.front=(q.front+1)%q.SIZE;
    el=q.item[q.front];
    System.out.println("Deleted element="+el);
  }
}
public void display()
{
  if(q.IsEmpty())
  {
    System.out.println("Queue empty");
  }
  else
  {
    int i;
    System.out.println("Queue elements are:");
    for(i=(q.front+1)%q.SIZE;i!=q.rear;i=(i+1)%q.SIZE)
    {
      System.out.print(q.item[i]+" ");
    }
    System.out.print(q.item[q.rear]+" ");
  }
}
public static void main(String args[])
{
  int choice;
  int el;
  Scanner sc=new Scanner(System.in);
  CQ_WithCount st=new CQ_WithCount();
  System.out.println("1:Enqueue");
  System.out.println("2:Dequeue");
  System.out.println("3:Display");
  do
```

```
    {
      System.out.println("Enter your choice");
      choice=sc.nextInt();
      switch(choice)
      {
        case 1:
          System.out.println("Enter data item to be inserted");
          el=sc.nextInt();
          st.Enqueue(el);
          break;
        case 2:
          st.Dequeue();
          break;
        case 3:
          st.display();
          break;
        default:
          System.out.println("Invalid choice Plz enter correct choice");
      }
    }while(choice<4);
  }
}
```

Output:

```
1: Enqueue
2: Dequeue
3: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
44
Enter your choice
1
Enter data item to be inserted
23
Enter your choice
1
Enter data item to be inserted
76
Enter your choice
1
Enter data item to be inserted
16
Enter your choice
1
Enter data item to be inserted
22
```

Queue full

Program no 22: Linked list implementation of circular queue in Java

```
import java.util.Scanner;
class CQNode
{
  public int info;
  public CQNode next;
  public CQNode()
  {
    info=0;
    next=null;
  }
}

class Linked_CQueue
{
  CQNode rear=new CQNode();
  CQNode front=new CQNode();
  public Linked_CQueue()
  {
    rear=null;
    front=null;
  }

  public void Enqueue(int item)
  {
    CQNode newnode=new CQNode();
    newnode.info=item;
    if(rear==null)
    {
      rear=front=newnode;
      newnode.next=newnode;
    }
    else{
      rear.next=newnode;
      rear=newnode;
      rear.next=front;
    }
}

public void Dequeue()
{
  if(front==null)
  {
    System.out.println("Empty queue");
  }
  else if(rear==front)
  {
    System.out.println("Deleted item="+front.info);
```

```
            rear=front=null;
        }
        else
        {
            CQNode temp=new CQNode();
            temp=front;
            front=front.next;
            rear.next=front;
            System.out.println("Deleted item="+temp.info);
        }
    }

    public void Display()
    {
        if(front==null)
        {
            System.out.println("Empty queue");
        }
        else
        {
            CQNode temp=new CQNode();
            temp=front;
            while(temp!=rear)
            {
                System.out.print(temp.info+" ");
                temp=temp.next;
            }
            System.out.print(rear.info+" ");
        }
    }

    public static void main(String args[])
    {
        int choice;
        int el;
        Scanner sc=new Scanner(System.in);
        Linked_CQueue st=new Linked_CQueue();
        System.out.println("1:Enqueue");
        System.out.println("2:Dequeue");
        System.out.println("3:Display");
        do
        {
            System.out.println("Enter your choice");
            choice=sc.nextInt();
            switch(choice)
            {
                case 1:
                    System.out.println("Enter data item to be inserted");
                    el=sc.nextInt();
                    st.Enqueue(el);
```

```
            break;
          case 2:
            st.Dequeue();
            break;
          case 3:
            st.Display();
            break;
          default:
            System.out.println("Invalid choice Plz enter correct choice");
        }
      }while(choice<4);
    }
  }
```

Program no 23: Array Implementation of ascending priority queue in Java

```
import java.util.Scanner;
class PQueue_Structure
{
  final int SIZE=10;
  public int item[]=new int[SIZE];
  public int rear,front;
  public PQueue_Structure()
  {
    rear=-1;
    front=0;
  }
  boolean IsFull()
  {
    return rear==SIZE-1;
  }
  boolean IsEmpty()
  {
    return rear<front;
  }
}
public class PriorityQueue
{
  PQueue_Structure q=new PQueue_Structure();
  public void Enqueue(int el)
  {
    if(!q.IsFull())
    {
      q.rear++;
      q.item[q.rear]=el;
    }
    else
    {
      System.out.println("Queue full");
    }
  }
```

```
    public void Dequeue()
    {
      int el, i, temp=0;
      el=q.item[q.front];
      if(q.IsEmpty())
      {
        System.out.println("Queue Empty");
      }
      else
      {
        for(i=q.front+1; i<=q.rear; i++)
        {
          if(el>q.item[i])
          {
            temp=i;
            el=q.item[i];
          }
        }
        for(i=temp; i<=q.rear; i++)
        {
          q.item[i]=q.item[i+1];
        }
        q.rear--;
        System.out.println("Deleted element is= "+el);
      }
    }

    public void display()
    {
      if(q.IsEmpty())
      {
        System.out.println("Queue empty");
      }
      else
      {
        int i;
        System.out.println("Queue elements are:");
        for(i=q.front;i<=q.rear;i++)
        {
          System.out.print(q.item[i]+" ");
        }
      }
    }
  public static void main(String args[])
  {
    int choice;
    int el;
    Scanner sc=new Scanner(System.in);
    PriorityQueue st=new PriorityQueue();
    System.out.println("1:Enqueue");
```

```
System.out.println("2:Dequeue");
System.out.println("3:Display");
do
{
  System.out.println("Enter your choice");
  choice=sc.nextInt();
  switch(choice)
  {
    case 1:
      System.out.println("Enter data item to be inserted");
      el=sc.nextInt();
      st.Enqueue(el);
      break;
    case 2:
      st.Dequeue();
      break;
    case 3:
      st.display();
      break;
    default:
      System.out.println("Invalid choice Plz enter correct choice");
  }
  }while(choice<4);
 }
}
```

Program no 24: Complete menu driven java program to perform various operations in Binary tree

```
import java.util.Scanner;
class BTNode
{
    BTNode left, right;
    int data;
    public BTNode()
    {
            left = null;
            right = null;
            data = 0;
    }
    public BTNode(int n)
    {
            left = null;
            right = null;
            data = n;
    }
    public void setLeft(BTNode n)   /* Function to set left node */
    {
            left = n;
    }
    public void setRight(BTNode n)  /* Function to set right node */
    {
```

```
            right = n;
        }
        public BTNode getLeft()  /* Function to get left node */
        {
            return left;
        }
        public BTNode getRight()  /* Function to get right node */
        {
            return right;
        }
        public void setData(int d)   /* Function to set data to node */
        {
            data = d;
        }
        public int getData()    /* Function to get data from node */
        {
            return data;
        }
    }

class BT
{
        private BTNode root;
        public BT()
        {
            root = null;
        }
        public boolean isEmpty()  /* Function to check if tree is empty */
        {
            return root == null;
        }
        public void insert(int data)     /* Functions to insert data */
        {
            root = insert(root, data);
        }
        private BTNode insert(BTNode node, int data)
        {
            if (node == null)
                node = new BTNode(data);
            else
            {
                if (node.getRight() == null)
                    node.right = insert(node.right, data);
                else
                    node.left = insert(node.left, data);
            }
            return node;
        }
        public int countNodes()     /* Function to count number of nodes */
```

```
{
        return countNodes(root);
}
private int countNodes(BTNode r)
/* Function to count number of nodes recursively */
{
        if (r == null)
                return 0;
        else
        {
                int l = 1;
                l += countNodes(r.getLeft());
                l += countNodes(r.getRight());
                return l;
        }
}
public boolean search(int val)  /* Function to search for an element */
{
        return search(root, val);
}
/* Function to search for an element recursively
private boolean search(BTNode r, int val)
{
        if (r.getData() == val)
                return true;
        if (r.getLeft() != null)
                if (search(r.getLeft(), val))
                        return true;
        if (r.getRight() != null)
                if (search(r.getRight(), val))
                        return true;
                return false;
}

public void inorder() /* Function for inorder traversal */
{
        inorder(root);
}
private void inorder(BTNode r)
{
        if (r != null)
        {
                inorder(r.getLeft());
                System.out.print(r.getData() +" ");
                inorder(r.getRight());
        }
}

public void preorder()  /* Function for preorder traversal */
{
```

```
                preorder(root);
        }
        private void preorder(BTNode r)
        {
                if (r != null)
                {
                        System.out.print(r.getData() +" ");
                        preorder(r.getLeft());
                        preorder(r.getRight());
                }
        }
        public void postorder()  /* Function for postorder traversal */
        {
                postorder(root);
        }
        private void postorder(BTNode r)
        {
                if (r != null)
                {
                        postorder(r.getLeft());
                        postorder(r.getRight());
                        System.out.print(r.getData() +" ");
                }
        }
}

public class BinaryTree   /* Class BinaryTree */
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        BT bt = new BT();
        System.out.println("Binary Tree Test\n");
        char ch;
        do
        {
            System.out.println("\nBinary Tree Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. search");
            System.out.println("3. count nodes");
            System.out.println("4. check empty");
            int choice = scan.nextInt();
            switch (choice)
            {
                case 1:
                    System.out.println("Enter integer element to insert");
                    bt.insert( scan.nextInt() );
                    break;
                case 2:
```

```
                    System.out.println("Enter integer element to search");
            System.out.println("Search result: "+ bt.search( scan.nextInt() ));
            break;
                case 3:
                    System.out.println("Nodes = "+ bt.countNodes());
                    break;
            case 4:
                System.out.println("Empty status = "+ bt.isEmpty());
                break;
            default:
                System.out.println("Wrong Entry \n ");
                break;
        }
        System.out.print("\nPost order : ");
        bt.postorder();
        System.out.print("\nPre order : ");
        bt.preorder();
        System.out.print("\nIn order : ");
        bt.inorder();
        System.out.println("\nDo you want to continue (Type y or n)");
        ch = scan.next().charAt(0);
    } while (ch == 'Y' || ch == 'y');
  }
}
```

Program no 25: Java program for inserting element in AVL Tree

```
class Node
{
    int key, height;
    Node left, right;
    Node(int d)
    {
        key = d;
        height = 1;
    }
}

class AVLTree
{
    static Node root;
    // A utility function to get height of the tree
    int height(Node N)
    {
        if (N == null)
        {
            return 0;
        }
        return N.height;
    }
```

```
// A utility function to get maximum of two integers
int max(int a, int b)
{
        return (a > b) ? a : b;
}
Node rightRotate(Node y)
{
        Node x = y.left;
        Node T2 = x.right;
        // Perform rotation
        x.right = y;
        y.left = T2;
        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;
        // Return new root
        return x;
}

Node leftRotate(Node x)
{
        Node y = x.right;
        Node T2 = y.left;
        // Perform rotation
        y.left = x;
        x.right = T2;
        // Update heights
        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;
        // Return new root
        return y;
}

// Get Balance factor of node N
int getBalance(Node N)
{
        if (N == null)
        {
        return 0;
        }
        return height(N.left) - height(N.right);
}

Node insert(Node node, int key)
{
        if (node == null)
        {
                return (new Node(key));
        }
        if (key < node.key)
```

```
            {
            node.left = insert(node.left, key);
            }
            else
            {
            node.right = insert(node.right, key);
            }
            node.height = max(height(node.left), height(node.right)) + 1;
             int balance = getBalance(node);
            if (balance > 1 && key < node.left.key)
            {
            return rightRotate(node);
            }
// Right Right Case
             if (balance < -1 && key > node.right.key)
            {
            return leftRotate(node);
            }
// Left Right Case
            if (balance > 1 && key > node.left.key)
            {
            node.left = leftRotate(node.left);
            return rightRotate(node);
            }
// Right Left Case
            if (balance < -1 && key < node.right.key)
            {
            node.right = rightRotate(node.right);
            return leftRotate(node);
            }
            return node;
        }

  void preOrder(Node node) {
    if (node != null) {
      System.out.print(node.key + " ");
      preOrder(node.left);
      preOrder(node.right);
    }
  }

public static void main(String[] args)
{
    AVLTree tree = new AVLTree();
    root = tree.insert(root, 5);
    root = tree.insert(root, 10);
    root = tree.insert(root, 21);
    root = tree.insert(root, 46);
    root = tree.insert(root, 50);
    root = tree.insert(root, 13);
```

```
    System.out.println("The preorder traversal of constructed tree is : ");
    tree.preOrder(root);
  }
}
```

Program no 26: java program to implement heap sort

```
import java.util.Scanner;
public class HeapSort
{
        private static int n;
        public static void sort(int arr[])
        {
                heapify(arr);
                for (int i = n; i > 0; i--)
                {
                        swap(arr,0, i);
                        n = n -1;
                        maxheap(arr, 0);
                }
        }
        public static void heapify(int arr[])
        {
                n = arr.length-1;
                for (int i = n /2; i >= 0; i--)
                    maxheap(arr, i);
        }
        public static void maxheap(int arr[], int i)
        {
                int left = 2*i ;
                int right = 2*i + 1;
                int max = i;
                if (left <= n && arr[left] > arr[i])
                    max = left;
                if (right <= n && arr[right] > arr[max])
                    max = right;
                if (max != i)
                {
                        swap(arr, i, max);
                        maxheap(arr, max);
                }
        }
        public static void swap(int arr[ ], int i, int j)
        {
                int tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
        }
        public static void main(String[ ] args)
        {
                Scanner scan = new Scanner (System.in );
```

```
            System.out.println("Heap Sort Test\n");
            int n, i;
            System.out.println("Enter number of integer elements");
            n = scan.nextInt();
            int arr[ ] = new int[n];
            System.out.println("\n Enter "+ n +" integer elements");
            for (i = 0; i < n; i++)
                arr[i] = scan.nextInt();
            sort(arr);
            System.out.println("\n Elements after sorting ");
            for (i = 0; i < n; i++)
                System.out.print(arr[i]+" ");
            System.out.println();
        }
    }
```

Output:

```
Heap Sort Test
Enter number of integer elements
20
Enter 20 integer elements
488 667 634 380 944 594 783 584 550 665 721 819 285 344 503 807 491 623 845 300
Elements after sorting
285 300 344 380 488 491 503 550 584 594 623 634 665 667 721 783 807 819 845 944
```

Program no 27: java program o implement splay tree

```
import java.util.Scanner;
class SplayNode
{
    SplayNode left, right, parent;
    int element;
    public SplayNode()
    {
        this(0, null, null, null);
    }
    public SplayNode(int ele)
    {
        this(ele, null, null, null);
    }
    public SplayNode(int ele, SplayNode left, SplayNode right, SplayNode parent)
    {
        this.left = left;
        this.right = right;
        this.parent = parent;
        this.element = ele;
    }
}
class SplayTree
{
```

```
private SplayNode root;
private int count = 0;
public SplayTree()
{
        root = null;
}
public boolean isEmpty()
{
        return root == null;
}

public void clear()
{
        root = null;
}
public void insert(int ele)
{
            SplayNode z = root;
            SplayNode p = null;
            while (z != null)
            {
                    p = z;
                    if (ele < p.element)
                            z = z.right;
                    else
                            z = z.left;
            }
            z = new SplayNode();
            z.element = ele;
            z.parent = p;
            if (p == null)
                    root = z;
            else if (ele < p.element)
                    p.right = z;
            else
                    p.left = z;
            Splay(z);
            count++;
}
public void makeLeftChildParent(SplayNode c, SplayNode p)
{
            if ((c == null) || (p == null) || (p.left != c) || (c.parent != p))
                    throw new RuntimeException("WRONG");
            if (p.parent != null)
            {
                    if (p == p.parent.left)
                            p.parent.left = c;
                    else
                            p.parent.right = c;
            }
```

```
        if (c.right != null)
                c.right.parent = p;
        c.parent = p.parent;
        p.parent = c;
        p.left = c.right;
        c.right = p;
}

public void makeRightChildParent(SplayNode c, SplayNode p)
{
        if ((c == null) || (p == null) || (p.right != c) || (c.parent != p))
                throw new RuntimeException("WRONG");
        if (p.parent != null)
        {
                if (p == p.parent.left)
                        p.parent.left = c;
                else
                        p.parent.right = c;
        }
        if (c.left != null)
                c.left.parent = p;
        c.parent = p.parent;
        p.parent = c;
        p.right = c.left;
        c.left = p;
}
private void Splay(SplayNode x)
{
        while (x.parent != null)
        {
                SplayNode Parent = x.parent;
                SplayNode GrandParent = Parent.parent;
                if (GrandParent == null)
                {
                        if (x == Parent.left)
                                makeLeftChildParent(x, Parent);
                        else
                                makeRightChildParent(x, Parent);
                }
                else
                {
                        if (x == Parent.left)
                        {
                                if (Parent == GrandParent.left)
                                {
                                        makeLeftChildParent(Parent,
GrandParent);
                                        makeLeftChildParent(x, Parent);
                                }
                                else
                                {
```

```
                    makeLeftChildParent(x, x.parent);
                    makeRightChildParent(x, x.parent);
                }
            }
            else
            {
                if (Parent == GrandParent.left)
                {
                    makeRightChildParent(x, x.parent);
                    makeLeftChildParent(x, x.parent);
                }
                else
                {
                makeRightChildParent(Parent, GrandParent);
                makeRightChildParent(x, Parent);
                }
            }
        }
    }
    root = x;
}
public void remove(int ele)
{
        SplayNode node = findNode(ele);
        remove(node);
}
private void remove(SplayNode node)
{
        if (node == null)
                return;
        Splay(node);
        if( (node.left != null) && (node.right !=null))
        {
                SplayNode min = node.left;
                while(min.right!=null)
                        min = min.right;
                min.right = node.right;
                node.right.parent = min;
                node.left.parent = null;
                root = node.left;
        }
        else if (node.right != null)
        {
                node.right.parent = null;
                root = node.right;
        }
        else if( node.left !=null)
        {
                node.left.parent = null;
                root = node.left;
```

```
            }
            else
            {
                root = null;
            }
            node.parent = null;
            node.left = null;
            node.right = null;
            node = null;
            count--;
}
public int countNodes()
{
        return count;
}
public boolean search(int val)
{
        return findNode(val) != null;
}
private SplayNode findNode(int ele)
{
            SplayNode z = root;
            while (z != null)
            {
                    if (ele < z.element)
                            z = z.right;
                    else if (ele > z.element)
                            z = z.left;
                    else
                            return z;
            }
            return null;
}
public void inorder()
{
        inorder(root);
}
private void inorder(SplayNode r)
{
            if (r != null)
            {
                    inorder(r.left);
                    System.out.print(r.element +" ");
                    inorder(r.right);
            }
}
public void preorder()
{
```

```
            preorder(root);
        }
        private void preorder(SplayNode r)
        {
                if (r != null)
                {
                        System.out.print(r.element +" ");
                        preorder(r.left);
                        preorder(r.right);
                }
        }

        public void postorder()
        {
            postorder(root);
        }
        private void postorder(SplayNode r)
        {
                if (r != null)
                {
                        postorder(r.left);
                        postorder(r.right);
                        System.out.print(r.element +" ");
                }
        }
}

public class SplayTreeTest
{
    public static void main(String[ ] args)
    {
            Scanner scan = new Scanner(System.in);
            SplayTree spt = new SplayTree();
            System.out.println("Splay Tree Test\n");
            char ch;
            do
            {
                    System.out.println("\nSplay Tree Operations\n");
                    System.out.println("1. insert ");
                    System.out.println("2. remove ");
                    System.out.println("3. search");
                    System.out.println("4. count nodes");
                    System.out.println("5. check empty");
                    System.out.println("6. clear tree");
                    int choice = scan.nextInt();
                    switch (choice)
                    {
                    case 1 :
                      System.out.println("Enter integer element to insert");
                      spt.insert( scan.nextInt() );
```

```
                    break;
                  case 2 :
                    System.out.println("Enter integer element to remove");
                    spt.remove( scan.nextInt() );
                    break;
                  case 3 :
                    System.out.println("Enter integer element to search");
                      System.out.println("Search result : "+ spt.search(
                     scan.nextInt() ));
                    break;
                  case 4 :
                    System.out.println("Nodes = "+ spt.countNodes());
                    break;
                  case 5 :
                    System.out.println("Empty status = "+ spt.isEmpty());
                    break;
                  case 6 :
                    System.out.println("\nTree Cleared");
                    spt.clear();
                    break;
                  default :
                    System.out.println("Wrong Entry \n ");
                    break;
                  }
                  System.out.print("\nPost order : ");
                  spt.postorder();
                  System.out.print("\nPre order : ");
                  spt.preorder();
                  System.out.print("\nIn order : ");
                  spt.inorder();
                   System.out.println("\nDo you want to continue (Type y or
                    n) \n");
                  ch = scan.next().charAt(0);
                } while (ch == 'Y' | | ch == 'y');
            }
        }
```

Program no 28: Complete program in java for BFS

```
import java.io.*;
import java.util.Scanner;
class Queue
{
  int items[]=new int[10];
  int front,rear;
  Queue()
  {
    front=0;
    rear=-1;
  }
  void Insert(int e)
```

```
            }
          }
        }
      }

      int getNumber()
      {
        int ne=0;
        Scanner in = new Scanner(System.in);
        ne=in.nextInt();
        return ne;
      }
    }
    public class BFS
    {
      public static void main(String args[])
      {
        Graph g=new Graph();
        g.createGraph();
      }
    }
```

Program no 29: Complete program in java for simulating DFS

```
import java.util.Scanner;
class Stack
{
  int stk[ ]=new int[10];
  int top;
  Stack()
  {
    top=-1;
  }
  void Push (int item)
  {
    if (top==9)
      System.out.println("Stack overflow");
    else
    stk[++top]=item;
  }
  boolean isEmpty()
  {
  if (top<0)
    return true;
  else
    return false;
  }
  int Pop()
  {
    if (isEmpty())
    {
```

```
      System.out.println("Stack underflow");
      return 0;
    }
    else
      return (stk[top--]);
  }
  void stackTop()
  {
    if(isEmpty())
      System.out.println("Stack underflow ");
    else
      System.out.println("Stack top is "+(stk[top]));
  }
  void Display()
  {
    System.out.println("Stack-->");
    for(int i=0;i<=top;i++)
      System.out.println(stk[i]);
  }
}
class Graph
{
  int MAXSIZE=51;
  int adj[][]=new int[MAXSIZE][MAXSIZE];
  int visited[]=new int [MAXSIZE];
  Stack s=new Stack();
  void createGraph()
  {
    int n,i,j,parent,adj_parent,initial_node;
    int ans=0,ans1=0;
    System.out.print("\nEnter total number elements in a Undirected Graph :");
    n=getNumber();
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
        adj[i][j]=0;
    for (int c=1;c<=50;c++)
      visited[c]=0;
    System.out.println("\nEnter graph structure for BFS  ");
    do
    {
      System.out.print("\nEnter parent node :");
      parent=getNumber();
      do
      {
        System.out.print("\nEnter adjacent node for node "+parent+ " : ");
        adj_parent=getNumber();
        adj[parent][adj_parent]=1;
        adj[adj_parent][parent]=1;
        System.out.print("\nContinue to add adjacent node for "+parent+"(1/0)?");
```

```
            ans1= getNumber();
        } while (ans1==1);
      System.out.print("\nContinue to add graph node?");
      ans= getNumber();
      }while (ans ==1);
      System.out.print("\nAdjacency matrix for your graph is :\n");
      for (i=1;i<=n;i++)
      {
        for (j=1;j<=n;j++)
        System.out.print(" "+adj[i][j]);
        System.out.print("\n");
      }
      System.out.println("\nYour Undirected Graph is :");
      for(i=1;i<=n;i++)
      {
        System.out.print("\nVertex "+i+"is connected to : ");
        for (j=1;j<=n;j++)
        {
          if (adj[i][j]==1)
            System.out.print(" "+j);
        }
      }
      System.out.println("\nEnter the initial node for BFS traversal:");
      initial_node=getNumber();
      DFS (initial_node, n);
    }
    void DFS (int initial_node,int n)
    {
      int u,i;
      s.top = -1;
      s.Push(initial_node);
      System.out.println("\nDFS traversal for given graph is : ");
      while(!s.isEmpty())
      {
        u=s.Pop();
        if(visited[u]==0)
        {
          System.out.print("\n"+u);
          visited[u]=1;
        }
        for (i=1;i<=n;i++)
        {
          if((adj[u][i]==1) && (visited[i]==0))
          {
            s.Push(u);
            visited[i]=1;
            System.out.print(" "+i);
            u = i;
          }
        }
```

```
  }
}
int getNumber()
{
  Scanner in = new Scanner(System.in);
  int ne=0;
  ne=in.nextInt();
  return ne;
}
}
class DFS
{
  public static void main(String args[])
  {
    Graph g=new Graph();
    g.createGraph();
  }
}
```

Program no 30: Complete Java program to simulate Kruskals algorithm fir finding MST of given graph

```
import java.util.*;
class krushkal
{
    public static void main(String ag[])
    {
    Scanner s=new Scanner(System.in);
        System.out.println("enter no of vertices");
        int v=s.nextInt();
        System.out.println("enter no of edges");
        int e=s.nextInt();
        int w[]=new int[e];
        int x[]=new int[e];
        int y[]=new int[e];
            for(int i=0;i<e;i++)
          {
                System.out.println("enter edges with weight");
            x[i]=s.nextInt();
            y[i]=s.nextInt();
            w[i]=s.nextInt();
              }
          for(int i=0;i<e;i++)
          for(int j=0;j<e-1;j++)
            {
            if(w[j]>w[j+1])
              {       int t=w[j];
                      w[j]=w[j+1];
                      w[j+1]=t;
                      t=x[j];
                      x[j]=x[j+1];
```

```
                    x[j+1]=t;
                    t=y[j];
                    y[j]=y[j+1];
                    y[j+1]=t;
                }}
            boolean b[]=new boolean[e];
            for(int i=0;i<e;i++)
              b[i]=false;
              int cost=0;
          for(int i=0;i<e;i++)
           {
                if(!(b[x[i]]&&b[y[i]]))
                  {b[x[i]]=true;
                      b[y[i]]=true;
                      cost+=w[i];}
          }
          System.out.println("the cost is "+cost)
          }
}
```

Program no 31: Complete java program to find MST from given graph by using Prim's algorithm

```
import java .util.*;
class Graph
{
        int g[][];
        int v,e;
        int d[],p[],visited[];
        void creategraph()
        {
                int a,b,w;
                Scanner s=new Scanner(System.in);
                System.out.println("Enter no. of vertices");
                v=s.nextInt();
                System.out.println("Enter no. of edges");
                e=s.nextInt();
                g=new int [v+1][v+1];
                for(int i=1;i<=v;i++)
                for(int j=1;j<=v;j++)
                g[i][j]=0;
                for(int i=1;i<=e;i++)
                {
                        System.out.println("Enter edge information");
                        a=s.nextInt();
                        b=s.nextInt();
                        System.out.println("Enter the wt of this edge");
                        w=s.nextInt();
                        g[a][b]=g[b][a]=w;
                }
        }
```

```
void callprim()
{
        visited=new int[v+1];
        d=new int[v+1];
        p=new int[v+1];
        for(int i=1;i<=v;i++)
                p[i]=visited[i]=0;
        for(int i=1;i<=v;i++)
                d[i]=32767;
        prim();
}
void prim()
{
        int c,current,mincost=0;
        current=1;
        visited[current]=1;
        d[current]=0;
        c=1;
        while(c!=v)
        {
                for(int i=1;i<=v;i++)
                {
                        if(g[current][i]!=0 && visited[i]!=1)
                                if(g[current][i]<d[i])
                                {
                                        d[i]=g[current][i];
                                        p[i]=current;
                                }
                }
                int min=32767;
                for(int i=1;i<=v;i++)
                {
                        if(visited[i]!=1 && d[i]<min)
                        {
                                min=d[i];
                                current=i;
                        }
                }
                visited[current]=1;
                c=c+1;
        }
        for(int i=1;i<=v;i++)
        mincost+=d[i];
        System.out.println("minimum cost= "+mincost);
}
}
Public class Prim
{

        public static void main(String args[])
        {
```

```
                Graph g=new Graph();
                g.creategraph();
                g.callprim();
        }
}
```

Program no 32: Complete program in C for Bubble sort

```
import java.util.Scanner;
class Bubble_Sort
{
        public static void main(String[] args)
        {
                inta[]=new int[100];
                intn,i,j,temp;
                    Scanner sc=new Scanner(System.in);
                System.out.println("Enter number of elements");
                    n=sc.nextInt();
                    System.out.println("Enter "+n+"numbers");
                    for(i=0;i<n;i++)
                    {
                        a[i]=sc.nextInt();
                    }
                    for(i=0;i<n-1;i++)
                    {
                      for(j=0;j<n-i-1;j++)
                      {
                        if(a[j+1]<a[j])
                        {
                          temp=a[j];
                          a[j]=a[j+1];
                          a[j+1]=temp;
                        }
                      }
                    }
                System.out.println("Elements in sorted order are");
                for(i=0;i<n;i++)
                        System.out.print(a[i]+" ");
        }
}
```

Program no 33: Complete Java program for Selection sort

```
import java.util.Scanner;
class Selection_sort
{
  public static void main(String[] args)
  {
        int a[ ]=new int[100];
        int n,i,j,temp, least,index;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter number of elements");
```

```
        n=sc.nextInt();
        System.out.println("Enter "+n+"numbers");
        for(i=0;i<n;i++)
        {
                a[i]=sc.nextInt();

        }
        for(i=0;i<n;i++)
        {
                least=a[i];
                index=i;
                for(j=i+1;j<n;j++)
                {
                if(a[j]<least)
                {
                                least=a[j];
                                index=j;
                        }
                }
                if(i!=index)
                {
                        temp=a[i];
                        a[i]=a[index];
                        a[index]=temp;
                }
        }
        System.out.println("Elements in sorted order are");
        for(i=0;i<n;i++)
        {
                System.out.print(a[i]+" ");
        }
    }
}
```

Program no 34: Complete program in C for Insertion sort

```
import java.util.Scanner;
class Insertion_Sort
{
 public static void main(String[] args)
 {
        int a[ ]=new int[100];
        int n, i, j, temp;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter number of elements");
        n=sc.nextInt();
        System.out.println("Enter "+n+"numbers");
        for(i=0;i<n;i++)
        {
                a[i]=sc.nextInt();
        }
        for(i=1;i<n;i++)
```

```
        {
                temp=a[i];
                j=i-1;
                while(j>=0&&(temp<a[j]))
                {
                a[j+1]=a[j];
                j--;
                }
                a[j+1]=temp;
        }
        System.out.println("Elements in sorted order are");
        for(i=0;i<n;i++)
        {
                System.out.print(a[i]+" ");
        }
    }
}
```

Program no 35: Complete Java program for Quick sort

```
package dsa_java;
import java.util.Scanner;
class QuickPartition
{
        int partition(int a[ ], int l, int r)
        {
                int x=l;
                int y=r;
                int p=a[l],temp;
                while(x<y)
                {
                 while(a[x]<=p)
                        x++;
                 while(a[y]>p)
                        y--;
                 if(x<y)
                 {
                        temp=a[x];
                        a[x]=a[y];
                        a[y]=temp;
                 }
        }
        a[l]=a[y];
        a[y]=p;
        return y;
    }
}

class Quick
{
  void QuickFun(int a[], int l, int r)
```

```
{
    int pivot;
    QuickPartition qp=new QuickPartition();
    if(l<r)
    {
        pivot=qp.partition(a, l, r);
        QuickFun(a, l, pivot-1);
        QuickFun(a, pivot+1, r);

    }

}

}

class Quick_Sort
{
public static void main(String[] args)
{
    int a[]=new int[100];
    int n, i, j;
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter number of elements");
    n=sc.nextInt();
    int l=0,r=n-1;
    System.out.println("Enter "+n+"numbers");
    for(i=0;i<n;i++)
    {
        a[i]=sc.nextInt();
    }
    Quick q=new Quick();
    q.QuickFun(a,l,r);
    System.out.println("Elements in sorted order are");
    for(i=0;i<n;i++)
    {
        System.out.print(a[i]+" ");
    }
  }
}
```

Program no 36: Complete Java program for Merge sort

```
package dsa_java;
import java.util.Scanner;
class Merge
{
    void Merge_Fun(int a[],intl,int r, int m)
    {
        int x=l;
        int k=l, i;
        intb[]=new int[100];
        int y=m;
        while(x<m && y<=r)
```

```
    {
            if(a[x]<a[y])
            {
                b[k]=a[x];
                    k++;
                    x++;
            }
            else
    {
                    b[k]=a[y];
                    k++;
                    y++;
            }
    }
    while(x<m)
    {
            b[k]=a[x];
            k++;
            x++;
    }
    while(y<=r)
    {
            b[k]=a[y];
            y++;
            k++;
    }
      for(i=l; i<=r; i++)
            a[i]=b[i];
      }
}

class Merging
{
   void MergingFun(int a[ ], int l, int r)
   {
        Merge m=new Merge();
        int mid;
        if(l<r)
        {
                mid=(l+r)/2;
                MergingFun(a, l, mid);
                MergingFun(a, mid+1, r);
                m.Merge_Fun(a, l,r,mid+1);
        }
   }
}

class Merge_Sort
{
 public static void main(String[] args)
```

```
    inta[ ]=new int[100];
    int n, i, j;
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter number of elements");
    n=sc.nextInt();
    int l=0,r=n-1;
    System.out.println("Enter "+n+"numbers");
    for(i=0;i<n;i++)
    {
        a[i]=sc.nextInt();
    }
    Merging m=new Merging();
    m.MergingFun(a,l,r);
    System.out.println("Elements in sorted order are");
    for(i=0;i<n;i++)
    {
        System.out.print(a[i]+" ");
    }
}
}
```

Program no 37: Complete Java program for Shell sort

```
class Shellsort
{
    public static void main(String args[ ])
    {
        int [ ] array = new int[ ] { 3, 2, 5, 4, 1 };
        int  k, i, j, increment, temp, number_of_elements = array.length;
        for (increment = number_of_elements / 2; increment > 0; increment /=2)
        {
            for (i = increment; i < number_of_elements; i++)
        {
                temp = array[i];
            for (j = i; j >= increment; j -= increment)
                {
                if (temp < array[j - increment])
                    {
                        array[j] = array[j - increment];
                }
                    else
                    {
                        break;
                }
            }
                array[j] = temp;
        }
        }
        System.out.println("After Sorting:");
        for (k = 0; k < 5; k++)
```

```
                {
                System.out.println(array[k]);
                }
        }
}
```

Program no 38: Complete java program for linear probing

```
import java.io.*;
import java.util.*;
import java.lang.*;
class DataItem
{
        public int iData;
        public DataItem(int ii)
        {
                iData = ii;
        }
}
class HashTable
{
        DataItem[ ] hashArray;
        int arraySize;
        DataItem nonItem;
        public HashTable(int size)
        {
                arraySize = size;
                hashArray = new DataItem[arraySize];
                nonItem = new DataItem(-1);
        }
        public void displayTable()
        {
                System.out.print("Table: ");
                for(int j=0; j<arraySize; j++)
                {
                        if(hashArray[j] != null)
                                System.out.print(hashArray[j].iData+ " ");
                        else
                                System.out.print("** ");
                }
                System.out.println(" ");
        }
        public int hashFunc(int key)
        {
                return key % arraySize;
        }

        public void insert(DataItem item)
        {
                int key = item.iData;
                int hashVal = hashFunc(key);
```

```
        while(hashArray[hashVal] != null && hashArray[hashVal].iData != -1)
        {
                ++hashVal;
                hashVal %= arraySize;
        }
        hashArray[hashVal] = item;
    }
    public DataItem delete(int key)
    {
        int hashVal = hashFunc(key);
        while(hashArray[hashVal] != null)
        {
                if(hashArray[hashVal].iData == key)
                {
                        DataItem temp = hashArray[hashVal];
                        hashArray[hashVal] = nonItem;
                        return temp;
                }
                ++hashVal;
                hashVal %= arraySize;
        }
        return null;
    }
    public DataItem find(int key)
    {
        int hashVal = hashFunc(key);
        while(hashArray[hashVal] != null)
        {
                if(hashArray[hashVal].iData == key)
                return hashArray[hashVal];
                ++hashVal;
                hashVal %= arraySize;
        }
        return null;
    }
}
class HashTableApp
{
    public static void main(String[ ] args)
    {
        DataItem aDataItem;
        int aKey, size, n, keysPerCell;
        putText("Enter size of hash table: ");
        size = getInt();
        putText("Enter initial number of items: ");
        n = getInt();
        keysPerCell = 10;
        HashTable theHashTable = new HashTable(size);
```

```
for(int j=0; j<n; j++)
{
        aKey = (int)(java.lang.Math.random() *
        keysPerCell * size);
        aDataItem = new DataItem(aKey);
        theHashTable.insert(aDataItem);
}
while(true)
{
        putText("Enter first letter of ");
        putText("show, insert, delete, or find: ");
        char choice = getChar();
        switch(choice)
        {
                case 's':
                        theHashTable.displayTable();
                        break;
                case 'i':
                        putText("Enter key value to insert: ");
                        aKey = getInt();
                        aDataItem = new DataItem(aKey);
                        theHashTable.insert(aDataItem);
                        break;
                case 'd':
                        putText("Enter key value to delete: ");
                        aKey = getInt();
                        theHashTable.delete(aKey);
                        break;
                case 'f':
                        putText("Enter key value to find: ");
                        aKey = getInt();
                        aDataItem = theHashTable.find(aKey);
                        if(aDataItem != null)
                        {
                                System.out.println("Found " + aKey);
                        }
                        else
                                System.out.println("Could not find " + aKey);
                        break;
                default:
                        putText("Invalid entry\n");
        }
}
}

public static void putText(String s)
{
        System.out.print(s);
        System.out.flush();
}
```

```
    public static String getString()
    {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            String s = br.readLine();
            return s;
    }
    public static char getChar() throws IOException
    {
            String s = getString();
            return s.charAt(0);
    }
    public static int getInt()
    {
            String s = getString();
            return Integer.parseInt(s);
    }
}
```

BIBLIOGRAPHY

1. Aho, J. Hopcroft, J.D. Ullman, *the Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1974.
2. Brassard, P. Bratley, *Algorithmic: Theory and Practice*, Englewood Cliffs, NJ: Prentice Hall, 1988.
3. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, New York: McGraw-Hill, 1991.
4. Knuth, *Fundamental Algorithms, volume 1 of The Art of Programming, 2nd ed.*, Menlo Park, CA: Addison-Wesley, 1973.
5. Knuth, *Seminumerical Algorithms volume 2 of The Art of Programming, 2nd ed.*, Menlo Park, CA: Addison-Wesley, 1981.
6. Knuth, *Sorting and Searching, volume 3 of The Art of Programming*, Menlo Park, CA: Addison-Wesley, 1973.
7. Manber, *Introduction to Algorithms: A Creative Approach*, Reading, MA: Addison-Wesley, 1989.
8. Sedgewick, *Algorithms, 2nd ed.*, Reading, MA: Addison-Wesley, 1989.
9. Yedidyah Langsam, Mosehe J. Augenstein, and Aaron M. Tenenbaum, *Data Structures using C and C++*, Prentice Hall of India
10. Data Structures and Algorithms Made Easy: Data Structures and **Algorithmic Puzzles**" by **Narasimha Karumanchi**.

Tribhuvan University
Institute of Science and Technology
Model Question Paper

Bachelor Level/ Second Year/ Third Semester/ Science — Full Marks: 60
Computer Science and Information Technology — Pass Marks: 24
(Data Structure and Algorithm) — Time: 3 hours.

Candidates are required to give their answers in their own words as for as practicable.
The figures in the margin indicate full marks.

Section A

Attempt any TWO questions. (2x10=20)

1. Describe using an example, how an arithmetic expression can be represented using a binary tree. Once represented, how can the expression be output in postfix notation?
2. Define stack as an ADT. Explain the condition that is to be checked for Push and Pop operations when stack is implemented using array?
3. Explain the advantages and disadvantages of representing a group of items as an array versus a linear linked list with suitable examples.

Section B

Attempt any EIGHT questions. (8x5=40)

4. Explain the difference between structure and union.
5. What is Big-O notation? Analyze the efficiency of quick sort.
6. Determine what the following recursive C function computes. Write an iterative function to accomplish the same purpose.

```
int func(int n)
{
        if (n==0)
            return (0);
        return (n+func(n-1));
}
```

7. Explain the concept of priority queue with an example.
8. Illustrate the sequential search with suitable example.
9. Write a non recursive depth-first traversal algorithm.
10. Write and explain the algorithm for Tower of Hanoi.
11. What is hashing? Explain the terms hash collision.
12. Explain why the straight selection sort is more efficient than the bubble sort.
13. Explain different types of binary tree.

***

TRIBHUVAN UNIVERSITY

FACULTY OF MANAGEMENT

Office of the Dean

BIM / Fourth Semester / IT 218 **Full Marks: 40**

Data Structure and Algorithm with Java **Time: 2 hrs.**

Candidates are required to answer all the questions in their own words as far as practicable.

Group "A"

1. Brief Answer Questions: ***[10*1 = 10]***

i. What is data structure?

ii. How doubly linked list differs from singly linked list?

iii. Differentiate between stack and queue. iv. What is splaying?

v. What do you mean by bucket Addressing?

vi. What is average case complexity of algorithm?

vii. What is adjacency matrix of graph?

viii. What is heap?

ix. What is tail recursion?

x. What is skip list?

Group "B"

Exercise Problems: ***[5* 4 = 20]***

2. Write Depth first traversal for graph.
3. Write a function in Java for insertion sorting.
4. Write a function in Java to insert a node in Binary Search tree.
5. Write a Java class to implement stack with push and pop functions.
6. Write a function in Java to delete element from a hash table.

Group "C"

Comprehensive Questions: ***[2*5 = 10]***

7. What are the advantages of B tree? Explain procedure to insert element in a B tree.
8. Explain quick sorting with an example.

***

KEC's B.Sc. CSIT Text Book Series

First Semester
- Introduction to Informatioin Technology
- C Programming
- Digital Logic
- Mathematics I
- Physics

Second Semester
- Discrete Structure
- Object Oriented Progamming
- Microprocessor
- Mathematics II
- Statistics I

Third Semester
- Data Structure and Algorithms
- Numerical Method
- Computer Architecture
- Computer Graphics
- Statistics II

Fourth Semester
- Theory of Computation
- Computer Networks
- Operating System
- Database Management System
- Artificial Intelligence

Fifth Semester
- Design and Analysis of Algorithms
- System Analysis and Design
- Cryptography
- Simulation and Modeling
- Web Technology
- Elective I

Sixth Semester
- Software Engineering
- Complier Design and Construction
- E-Governance
- NET Centric Computing
- Technical Writing
- Elective II

Seventh Semester
- Advanced Jave Programming
- Data Warehousing and Data Mining
- Principles of Management
- Project Work
- Elective III

Eighth Semester
- Advanced Database
- Internship
- Elective IV
- Elective V

KEC
Publication and Distribution Pvt. Ltd.
Kathmandu, Nepal, Tel.: 01-4168301, 01-4241777
E-mail: kecpublication14@gmail.com