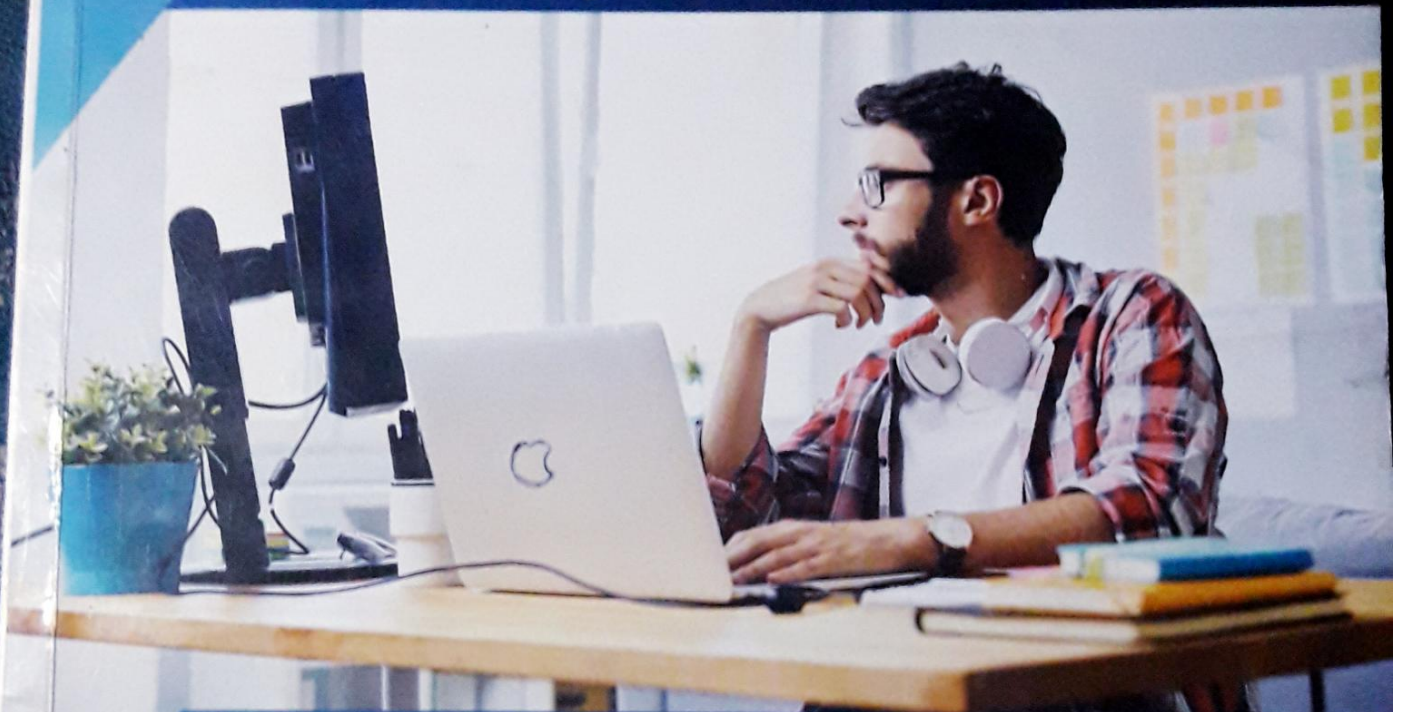


**Sarthak**

According to the New Revised Syllabus of UPBTE



# Data Structure Using 'C'

Satish Gupta

**Jai Prakash Nath Publications**  
Meerut



# UPBTE SYLLABUS

## Data Structure Using 'C'

**Rationale :** For solution of different problems 'C' is a very powerful high level language. It is widely used in research and engineering problems. A software technician aware of this language will be useful for working in computer environment.

### TOPIC-WISE DISTRIBUTION OF PERIODS

S. No.	Units	Coverage Time		
		L	T	P
1.	Basic Concepts	6	—	—
2.	Lists	12	—	—
3.	Stacks and Queues	10	—	—
4.	Sorting & Merging	10	—	—
5.	Tables	10	—	—
6.	Tree	12	—	—
7.	Graphs	10	—	—
		<b>70</b>	<b>—</b>	<b>70</b>

### DETAILED CONTENTS

#### 1. Basic Concepts

- 1.1 Problem solving concept, top down and bottom up design, Structured programming, concept of data type, variable and constants, Introduction to Data Structure (Linear, Non Linear, Primitive, Non Primitive), Concept of data structure (Array, Linked List Stack, Queue, Trees, Graphs).
- 1.2 Array : Concept of arrays, single dimensional array, Two Dimensional array Representation of two dimensional array (Base address, L.B., U.B.), Operation of arrays with algorithms (Searching, traversing, inserting, deleting).

#### 2. Lists

Introduction to linked list and double linked list, Representation of linked lists in memory, comparison between linked list and Array, Traversing a link list, Searching a link list, Insertion and deletion into linked list (Ar



first node, Specified Position, Last node), Application of link list, Doubly linked lists, Traversing a doubly link lists, Insertion and deletion into doubly Link list.

### **3. Stacks And Queues**

Introduction to stacks, representation of stacks with array and link List, Implementation of stacks, Application of stacks (Polish Notations, converting infix to post fix notation, evaluation of Post fix notation, tower of Hanoi), Recursion : Concept and Comparison between recursion and iteration, Introduction to queues, Implementation of queues, Circular queues, De-queues.

### **4. Sorting Algorithms**

Introduction, Search algorithm (Linear and Binary), Concept of sorting. Insertion sorts, Bubble sort, Quicksort, Mergesort, Heapsort.

### **5. Tables**

Searching sequential tables, Hash tables and Symbol tables, Heaps.

### **6. Trees**

Concept of Binary Trees (Complete, Extended Binary Tree), Concept of representation of Binary Tree, Concept of balance Binary Tree, Traversing Binary Tree (Pre order, Post order and In Order), Searching Inserting and deleting in binary search trees.

### **7. Graphs**

Depths-first-search.



# CONTENTS

Chapter	Page Nos.
1 Basic Concepts	1-48
2 Lists	49-76
3 Stacks and Queues	77-114
4 Sorting Algorithms	115-143
5 Tables	144-158
6 Trees	159-185
7 Graphs	186-209
Experiments/Practicals	210-224





## Unit

# 1

## Basic Concepts

**प्रश्न 1. Variable क्या है? ये कितने प्रकार के होते हैं? या विभिन्न प्रकार के Variables का Example के द्वारा वर्णन करें।**

**उत्तर—**Computer पर कोई भी Operation करने के लिए कुछ Data की requirement होती है। उस Data को Computer के Memory में Save किया जाता है जब कि Memory अलग-अलग Cells में बटी होती है और हर Cell का अपना अलग-अलग Address होता है। इस Address से Data को Access (प्राप्त) करने के लिए या Store करने के लिए उन Datas को Variable या Constant के नाम से जाना जाता है।

### Variable

Variable एक 'नाम' या Entity को कहा जाता है जो कि Programming में एक बार में केवल एक value को store करता है तथा साथ ही इसकी value, Program के Execution/run time पर बदला (Change) भी जा सकता है।

### Declaration of Variable

Program में किसी Variable को use करना है तो सबसे पहले इसे बनाते हैं, जिसमें यह बताया जाता है कि variable किस प्रकार का Data type है (int, char, void, float, etc). और Memory में कितना जगह (Address) लेगा। इसके Syntax को निम्न प्रकार से represent करेंगे—

Datatype Variablename ;

इस syntax के आधार पर किसी भी Variable को निम्न प्रकार Declare किया जा सकता है—

Example :

int x ;

यहाँ पर x एक integer (int) प्रकार का Variable है और यह केवल और केवल Integer Value को ही store करेगा।

### Initialization of Variable

किसी Variable के बनने (Declaration) के बाद इसको कोई एक value assign (देने) करने की प्रक्रिया को (Initialization) कहा जाता है। किसी variable की value को दो प्रकार से Declare किया जाता है—

1. Static Initialization
2. Dynamic Initialization



**1. Static Initialization :** इस प्रकार का Initialization Program में Variable Declaration के साथ ही इसकी कोई ना कोई Value Assign कर दी जाती है। इस प्रकार का initialization Static, Initialization कहलाता है। जैसे—

```
void main ()
{
    int x ; // यहाँ पर x नाम का, और Integer Type का variable बनाया गया है।
    x = 10 ; // यहाँ पर x की value 10, Initialize/Assign किया गया है।
```

**Note :** इस प्रकार का initialization पूरे Program के दौरान कभी Change नहीं किया जा सकता।

**2. Dynamic initialization :** इस प्रकार के Initialization में किसी variable की value Program के Execution/run कराने के दौरान (time) दिया है इस प्रकार से initialize किये गये value को change किया जा सकता है। जैसे कि—

```
void main ( )
{
    int a, b, c; // यहाँ पर केवल Variable a, b, c Declare किया गया है।
    // इनको कोई Value initialize नहीं किया गया है।
    printf ("Enter value of a & b"); // यहाँ पर a और b की value user से पूछा जा रहा है।
    scanf ("%d %d", &a, &b);
    // user से लिये गये value को Memory में Save करने के लिए scanf( ) का प्रयोग किया जाता है।
    c = a + b; // यहाँ पर a और b की value को c में Assign किया गया है।
    printf ("%d", c); // c की value को जानने के लिए।
}
```

**Note :** इस Program में a, b और c की value नहीं दिया गया है, यह Program जब Execute होगा इस समय user से पूछा जाता है कि a और b की value क्या होगा। इस प्रकार a और b की value हर बार change किया जा सकता है।

## Types of Variable

Program में Variable को use करने के लिए दो प्रकार के variables का use किया जाता है—

- (i) Local Variable
- (ii) Global Variable

**1. Local Variable :** जब कोई Variable किसी Function (Main ()) के (केवल और केवल) body के अन्दर ही प्रयोग (use) किया जा सकता हो, बाहर नहीं, तो इस प्रकार का Variable, Local Variable कहलाता है। जैसे कि—

```
void main ( )
{
    int a, b, c;
```



```
int a, b, c;
a = 10, b = 30;
c = a + b;
printf ("%d", c);
```

यहाँ पर variable  $a$ ,  $b$  और  $c$ , जो कि `main()` के body में है इसलिए ये local variable कहलायेगे और इसका use, main function के बाहर नहीं किया जा सकता।

**2. Global variable :** वो variable जो `main()` के पहले Declare किया जाता है और body के अन्दर और बाहर दोनों जगहों पर प्रयोग (use) किया जा सकता है, वह Global Variable कहलाता है। जैसे कि—

```
int c;
void main ()
{
    int a, b;
    a = 100, b = 200;
    c = a + b;
    printf ("%d", c);
}
```

यहाँ पर  $a$  और  $b$  `main()` function के body में है और variable  $c$  `main()` function से पहले है इसलिए  $a$  और  $b$  local variable और  $c$  global variable कहलायेगा।

**Note :** Global Variable हमेशा `main()` से पहले Declare किया जाता है।

## Rules for Naming Variable (Variable बनाने के नियम)

कुछ जरूरी बातें जो variable declare करते समय ध्यान रखना चाहिए, जैसे कि—

1. Variable बनाने के लिए letters, digits और एक special symbol underscore (`_`) का use किया जाता है।
2. Variable name का पहला अक्षर या तो letter से शुरू हो या Underscore (`_`) से। Digit से start नहीं होगा।
3. Variable case sensitive होता है। इसलिए अगर variable जिस case (upper case या lower case) में बनाया गया है इसी तरह Access/use करना चाहिए।
4. Variable को Datatype की जगह और Datatype को Variable की जगह use नहीं किया जा सकता।

For example,

### Valid Variable Name

```
int A_9;      float _X;
char A;       int A9_;
```

### Invalid Variable name

```
int 9A;       char 9A;
float 9_x;    int 9x_;
```

**प्रश्न 2. Data type क्या होता है और ये कितने प्रकार के होते हैं? विभिन्न प्रकार के Data types का Example के साथ वर्णन करें।**



**उत्तर—**Programming language 'C' में, हर Variable के साथ एक Data type जरूर होता है जो यह बताता है कि वह जिस Variable के साथ है वह Variable किस प्रकार का value store करेगा और इसके लिए कितना memory address लेगा (occupy) यह किसी भी variable के पूरे विवरण (specification) information को represent करता है। Datatype को निम्न प्रकार से declare किया जाता है—

**Syntax :** Datatype Variablename ;

**Example :** int x ;

यहाँ पर x एक variable है जो कि int (integer) type का है इससे compiler को यह पता चलता है कि x केवल और केवल integer value को store करेगा और इस value को store करने के लिए 2 bytes का memory occupy करेगा।

## Types of data type

'C' Programming language में कई सारे Data type हैं जिनकी विशेषताएँ (qualities) अलग-अलग हैं। इन सभी Data types को तीन categories में बाँटा गया है—

1. Primary (Built-in)/Fundamental Data type.
2. Derived Data type.
3. User defined data type

**1. Primary (Built-in)/Fundamental datatype :** 'C' Compiler पाँच प्रकार के Primany data types को support करता है—

(a) int (2) char (3) float (4) double (5) void

ये Data type किस प्रकार की Values Store करेंगे, कितना memory occupy (लेंगे), उनका Range क्या होगा, और इनका Access specifier को किस तरह represent करेंगे।

Data type	Value	Size	Range	Access specifier
int	numeric	2	-32768 to 32767	%d, %i
char	character	1	-128 to 127	%c
float	decimal	4	$3.4E-38$ to $3.4E+38$	%f
double	decimal	8	$1.7E-308$ to $1.7E+308$	%lf
void (null)	—	—	—	—

**Note :** void data type को Null type के नाम से भी जाना जाता है इसका use variable के साथ नहीं किया जाता क्योंकि यह कोई memory occupy नहीं करता। यह केवल और केवल Function के साथ use किया जाता है। जो कि कोई Value return नहीं करता।

**2. Derived Data type :** ये वे Data types होते हैं जो कि Fundamental/Built-in data type से मिलकर बने होते हैं। जैसे कि—

- (a) Array (b) Structure (c) Union (d) Function etc.

**(A) Array :** Array, एक ही तरह के कई सारे elements/values का एक समूह (collection) है। इसको इस प्रकार से represent किया जाता है—



**Syntax**

Datatype Arrayname [Size] :

Example int x [5] :

यहाँ पर  $x$  एक Array है जिसका size 5 है मतलब यह कि  $x$ , 5 element को store करेगा, जो कि सभी element integer types के होंगे।

(B) **Structure** : यह कई अलग-अलग Data type के elements का एक collection है। इसको इस प्रकार represent करेंगे—

```
struct tagname
```

```
{
```

```
    element 1;
```

```
    element 2;
```

```
    element n;
```

```
};
```

Structure को represent करने के लिए struct keyword का use किया जाता है। जैसे कि—

```
struct Book
```

```
{
```

```
    char Author_name [25];
```

```
    int pages;
```

```
    float Price;
```

```
};
```

(C) **Union** : union भी कई सारे elements (जो कि अलग-अलग Data types के हैं) का एक collection होता है। इसको represent करने के लिए union keyword का use किया जाता है।

```
union Book
```

```
{
```

```
    char Author_name [25];
```

```
    int pages ;
```

```
    float Price ;
```

```
};
```

(D) **Function** : function, कुछ elements का एक collection होता है जो किसी एक special काम को करने के लिए बनाया जाता है। इसको इस प्रकार से represent किया जाता है—

**Syntax :**

```
Returntype Function_name (Parameter List)
```

```
{
```

```
    Instruction (s) :
```

```
}
```

**Example :**

```
int sum (int x, int y)
```



```
{
    return (x + y);
}
```

**3. User defined datatype :** User के द्वारा अपने काम को पूरा करने के लिए, एक नया बनाया गया Data type, User defined datatype कहलाता है। जैसे—

1. typedef

2. enum

**1. typedef :** किसी identifier को, किसी primary/fundamental data type में बदलने के लिए typedef, keyword का use किया जाता है।

या

typedef एक keyword है जिसके द्वारा किसी identifier को datatype में बदला जा सकता है। इसके लिए इस प्रकार syntax लिखें—

```
typedef datatype identifier;
```

यहाँ पर Datatype कोई Fundamental/derived datatype है और identifier, जिसको Data type में user को typedef के द्वारा convert करना है।

**Example :** typedef int marks;

अब marks एक identifier नहीं बल्कि एक Data type बन जायेगा : इसलिए marks को अब int की जगह use किया जा सकता है—

**Example :** marks mark 1, mark 2;

यहाँ पर marks एक int type का data type है और mark 1, mark 2 integer type के variables हैं।

**2. enum :** enum एक keyword है जो कि enumeration का short form है। enumeration एक Integer Values का Collection है। इसे निम्न प्रकार से represent किया जाता है—

**Syntax :** enum identifier { value 1, value 2, ..., value n};

यहाँ पर identifier एक user defined enumerated data type है जिसका प्रयोग variable की value को एक braces में declare करने के लिए किया जाता है।

**declaration**

```
enum day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

**Example :** enum day day 1, day 2, day 3;

```
day 1 = Monday
```

```
day 2 = Tuesday
```

```
day 3 = Sunday || An error occurs.
```

enum की first name की value शून्य (0), अगले (second) name की value एक (1), इसी तरह आगे की हर name की value एक-एक करके बढ़ेगी।

अगर हम enum के name को कोई value नहीं देते हैं तो पहले name की value शून्य (0) by default (स्वतः) ही assign हो जाएगी। जैसे कि—

```
enum day { Monday, Tuesday, ..... Saturday};
```



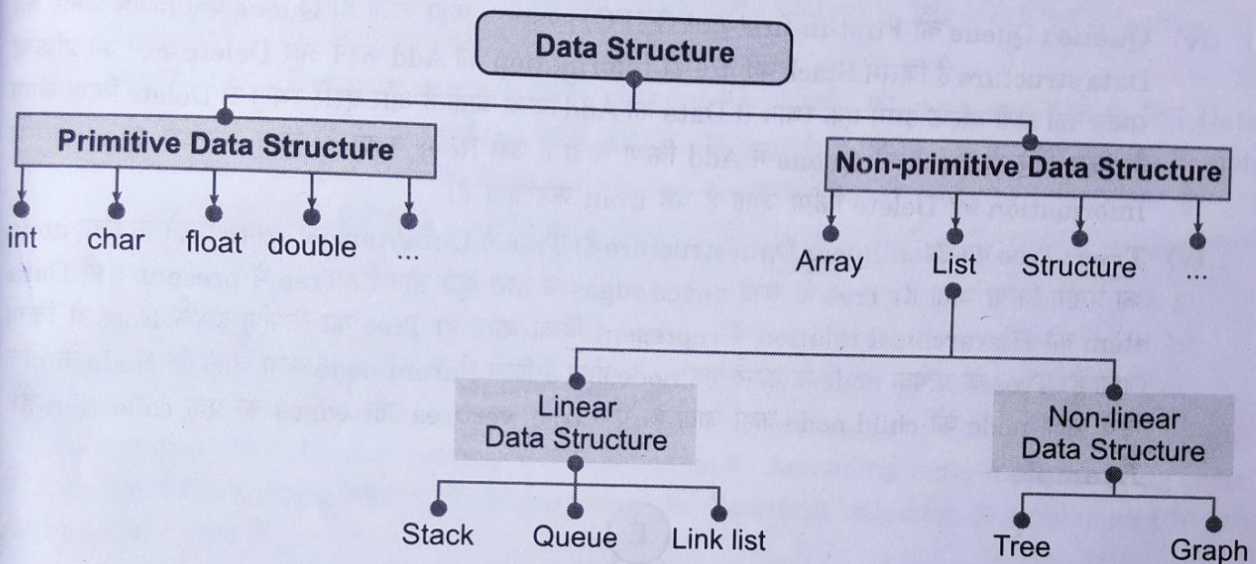
यहाँ पर Monday की value 0 तथा Tuesday की value 1 और Saturday की value 5 होगी।

**प्रश्न 3. Data Structure क्या होता है? इसके classificaton को समझाइये।**

**उत्तर—**किसी भी Program में Data को विभिन्न प्रकार से Arrange किया जा सकता है। किसी भी Data का logical या Arithmetical Representation ही Data Structure कहलाता है। किसी Data Representation को choose करना दो concepts पर निर्भर करता है—

1. Data इतना Arranged होना चाहिए कि वह Actual Relation को प्रदर्शित (Represent) कर सके।
2. वह इतना Easy होना चाहिए कि आवश्यकता पड़ने पर इसे आसानी से समझा जा सके।

### Classification of Data Structure :



#### Primitive Data Structure

ऐसे Data type जो Directly किसी Machine से communicate करते हैं Primitive Data Structure कहलाते हैं।

जैसे कि— int, char, float, double, etc.

#### Non-primitive Data Structure

ऐसे Data type जो indirectly किसी Machine से communication करते हैं Non-primitive Data Structure कहलाते हैं।

जैसे कि— Array, Structure, Function, list, etc.

#### Linear Data Structure

यह एक ऐसा Data Structure है जिसमें Elements Sequence में store रहते हैं तथा इन पर Operations भी sequence के रूप में ही होता है। Linear Data Structure के उदाहरण निम्न हैं—

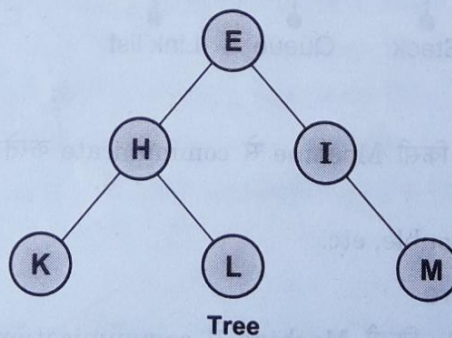
Array, linked list, Stack तथा queue.

**I. Array :** Array, same types of data का एक collection होता है। Array के elements को Index के रूप में memory में represent किया जाता है।



- II. Linked list :** link list, Data item का एक linear collection है जिसे node कहते हैं। Linear order को Pointers के द्वारा maintain किया जाता है। हर node दो भागों में divide होता है।
- III. Stack :** Stack को Last-in-first-out (LIFO) system भी कहा जाता है। यह एक liner link list है जिसमें Insertion और Deletion एक ही end से किया जाता है जिसे Top कहते हैं। Stack में Data को Insert और Delete करने के लिए केवल दो ही Function हैं—
- (a) **Push :** Stack में Data को Store/Instert/Add करने के लिए Push function का प्रयोग किया जाता है।
  - (b) **Pop :** Stack से किसी Data को Delete/Remove करने के लिए Pop function का प्रयोग किया जाता है।
- (iv) **Queue :** Queue को First-in-first-out (FIFO) system कहा जाता है। Queue एक विशेष प्रकार का Data structure है जिसमें Stack की तरह ही Information को Add करने और Delete करने की प्रक्रिया समान नहीं होती बल्कि इसमें एक स्थान से Data को Add किया जाता है और दूसरे स्थान से Delete किया जाता है। जिस ओर से Data को Queue में Add किया जाता है उसे Rear कहते हैं। जिस स्थान से Queue में से Information को Delete किया जाता है वह front कहलाता है।
- (v) **Tree :** Tree एक Non-linear Data structure है। Tree में Data/item को store करने के लिए node का प्रयोग किया जाता है। Tree के सभी nodes edges के द्वारा जुड़े होते हैं। Tree में present सभी Data item को Hierarchical relation में represent किया जाता है। Tree का निर्माण इसके Root से किया जाता है। Tree के प्रत्येक node के ऊपर एक node होता है जिसे Parent node कहा जाता है। Node के ठीक नीचे वाली node को child node कहा जाता है। एक tree, vertices और edges का एक collection है।

**Example :**



**Graph :** कभी-कभी Data Structure Elements के मध्य relationship को represent बिना Hierarchical relation के भी किया जा सकता है। ऐसा Data Structure जो किसी विशेष Relationship को represent करता है Graph Data structure कहलाता है। Set of nodes और set of edges को Graph कहते हैं। Graph दो प्रकार के होते हैं—

1. Direct Graph

2. In-direct Graph

**प्रश्न 4. Data Structure पर होने वाले operation को समझाइए।**

उत्तर—किसी भी Data Structure पर Different operation को perform किया जा सकता है। Data Structure पर मुख्य रूप से 6 प्रकार के operations perform किये जाते हैं—

1. Insertion

4. Merging



2. Searching

3. Sorting

5. Traversing

6. Deletion

**1. Insertion :** किसी दिये गये Data Structure में किसी नये Data को Add करने की प्रक्रिया को Insertion कहा जाता है। नये element को किसी भी जगह Insert किया जा सकता है। अगर Data को Data structure में उपस्थित element के बाद Insert किया जाता है तो यह special operation append कहलाता है। एक ऐसा situation भी आता है जब Data structure में कोई और Element insert नहीं किया जा सकता, इस situation को overflow कहा जाता है क्योंकि Data Structure पहले से ही Full है।

**Example :** यदि एक Array है जिसकी size 30 का है इसमें 25 element पहले से ही हैं अब नये element को इस Array list पर एक Position पर जोड़ने की प्रक्रिया (insertion operation) कहलाता है।

**2. Searching :** Data structure में किसी element की उपस्थिति को represent करने के लिए Data structure पर search operation perform किया जाता है। यह search या तो linear होगा या Binary Search होगा। Key element के आधार में किसी Data structure में Search operation perform किया जाता है।

**Example :** माना कि एक Array जिसका size 5 है इसमें 10, 9, 20, 11, 6 element हैं इनमें से element 20 को search करना है तो सबसे पहले Array के 1st element से 20 को compare करते हैं और last तक किया जाता है। यदि 20 उस Array में मिल जाता है तो इस Operation को successful और ना मिलने पर unsuccessful कहा जाता है।

**3. Sorting :** किसी Data Structure के सभी elements को Ascending order या Descending order में Arrange करना sorting कहलाता है। इस operation को Insertion, selection या bubble sort के द्वारा perform किया जाता है।

**Example :** यदि एक Array है जिसका size 10 है और इसमें 10, 20, 8, 7, 28, 26, 40, 50, 11, 18 elements हैं जो कि किसी भी order में Arrange नहीं हैं। इस Array को Ascending order में Arrange किया जाएगा तो यह order 7, 8, 10, 11, 18, 20, 26, 28, 40, 50 प्राप्त होगा।

**4. Merging :** किन्हीं दो lists A और B जिनका size M और N हैं, जो कि एक ही प्रकार के हैं ऐसे lists को जोड़ने (Merge करने) के लिए एक तीसरे list C की जरूरत होगी जिसका size  $M + N$  elements का होगा। इस प्रक्रिया को करने के लिए Merging operation की जरूरत पड़ेगी।

**Example :** माना कि दो lists, जिसमें पहले list में 6 elements, 10, 20, 30, 40, 50 और 60 elements है जब कि दूसरे list में 5 elements 8, 15, 18, 25, 35 हैं। ये दोनों lists Ascending order में हैं। इनको merge करने के लिए एक ऐसा List चाहिए जिसका size 11 हो। दोनों lists को merge करने पर order 8, 10, 15, 18, 20, 25, 30, 35, 40, 50, 60 मिलेगा।

**5. Traversing :** किसी Data Structure में उपस्थित सभी elements को Display करने के लिए, हर Data elements को कम से कम एक बार Visit करना पड़ता है। इस प्रक्रिया को traversing कहते हैं।



**Example :** किसी Data Structure में उपस्थित elements को count करना हो तो हर element को एक बार Visit जरूर करना पड़ेगा, यही प्रक्रिया Traversing कहलाती है।

**6. Deletion :** किसी Data Structure से किसी एक element को remove करने की प्रक्रिया को Deletion कहते हैं। Data Structure में एक ऐसा situation भी आता है जब इस समय कोई और Delete operation नहीं किया जा सकता। इस Situation को Underflow कहते हैं।

**Example :** किसी Array में 10 elements 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 हैं इनमें से 70 element को remove करना है तब Array में 9 elements होंगे जो इस प्रकार हैं—10, 20, 30, 40, 50, 60, 80, 90, 100.

## Data Structure

Data may be organized in many different ways : the logical or mathematical Model of a particular organization of data is called Data Structure. The choice of a particular data model depends on two considerations. First, it must be rich enough in structure to mirror the actual relationship of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary.

Algorithm + Data structure = Program

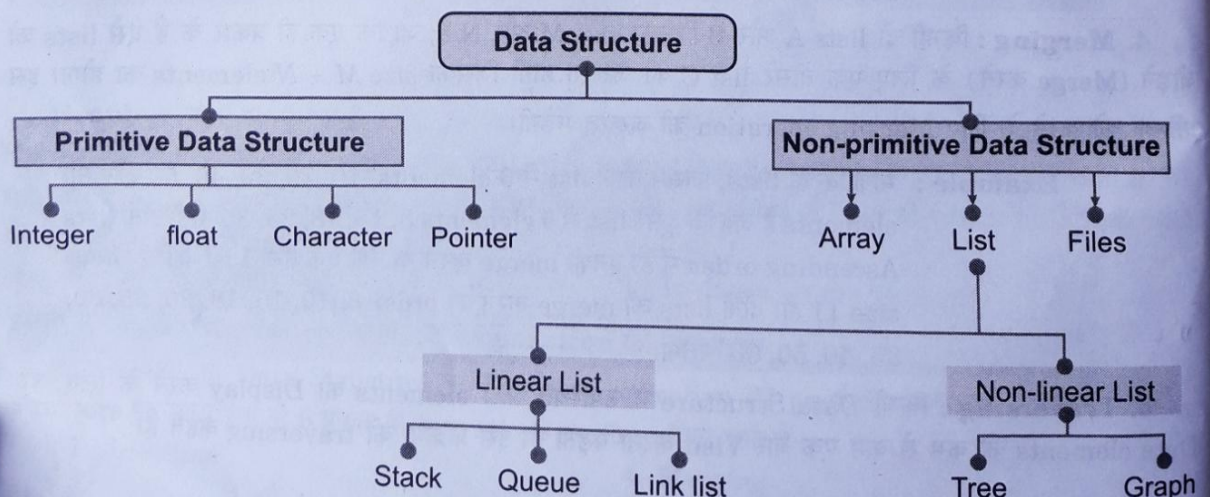
You know that an algorithm is a step-by-step procedure to solve a particular function. That is, it is a set of instruction written to carry out certain tasks and the data structure is the way of organizing the data with their logical relationship retained.

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

## Classification of Data Structures

Data structures are normally divided into two broad categories :

1. Primitive Data structure
2. Non-primitive Data structure





6. **Merging**: It is the process of combining the element of two sorted data structures into a single sorted data structure. Both the structures to be merged should be similar.

### Analysis of an Algorithm

Algorithm का विश्लेषण, किसी Algo के ठीक होने को और इसको लागू करने की प्रक्रिया तथा Algorithm को पूरा होने में लगे समय का प्रबंधन करता है।

किसी दिए हुए Algo के 3 प्रकार के विश्लेषण हैं और उनके system के परिमाण हैं—

1. Worst Case पूर्ण समय
2. Average Case पूर्ण समय
3. Best Case पूर्ण समय

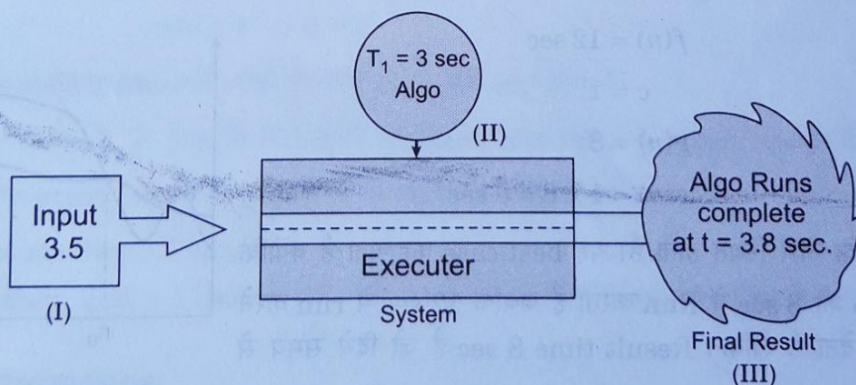
किसी Algo के विश्लेषण की अवस्था को जानने के तीन (3) अवयव होते हैं—

1. Input पूर्ण समय।
2. Program के step के आधार पर सम्भावित समय जो Algo लेगी:
3. Program का run किया गया समय जो वास्तविक समय इसे system पर चलाने पर लगा (Final मान)।

Input मान = 3.5 सेकेण्ड

Algo के step के अनुसार सम्भावित समय = 3 सेकेण्ड

Run किया गया समय (अंतिम समय) = 3.8 सेकेण्ड।



### Asymptotic Analysis

यह एक गणितीय विश्लेषण है जो किसी Algorithm पर किया जाता है। इसमें किसी भी Run किये गये Program पर उचित steps की संख्या का आकलन भी किया जाता है।

ये तीन (3) प्रकार के होते हैं—

1. Bigoh (O) Notation
2. Omega ( $\Omega$ ) Notation
3. Theta ( $\Theta$ ) Notation

61



### 1. Bigoh (O) Notation :

फलन  $f(n) = O(g(n))$  होगा जब वहाँ पर एक +ve constant (अचर)  $c$  और एक boundary मान  $n_0$ .

माना Algo को run करने में Actual Time 10 sec. लगता है और system इसे run कराने में किसी Constant value के साथ like  $cg(n)$ ,  $f(n)$  के ऊपर स्थित होती है जिसे upper bound कहते हैं।

इस stage को Algo का upper bound बोलते हैं, यह Worst case है।

$$f(n) = 10 \text{ sec.}$$

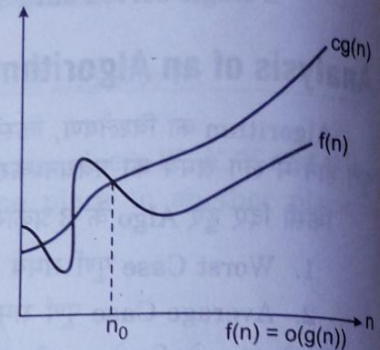
$$g(n) = 8 \text{ sec.}$$

$$c = 2$$

अब

$$f(n) = 10 \text{ sec.}$$

$$cg(n) = 2 \times 8 = 16 \text{ sec.}$$



### 2. Omega (Ω) Notation

Big omega ( $\Omega$ ) notation Lower bound है क्योंकि—

माना

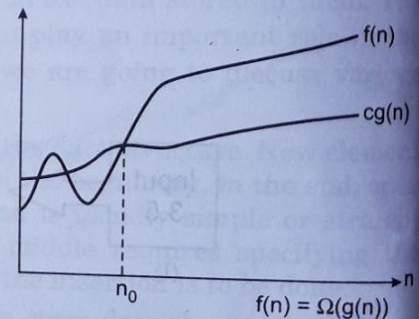
$$f(n) = 12 \text{ sec}$$

$$c = 1$$

$$g(n) = 8$$

$$cg(n) = 1 \times 8 = 8 \text{ sec}$$

$cg(n)$ ,  $f(n)$  के नीचे स्थित होता है। यह best case कहलाता है क्योंकि system उस Algo को 8 sec में Run करता है जबकि 12 sec में run करने का expectation देता है लेकिन Result time 8 sec है जो दिये समय से काफी पहले है।



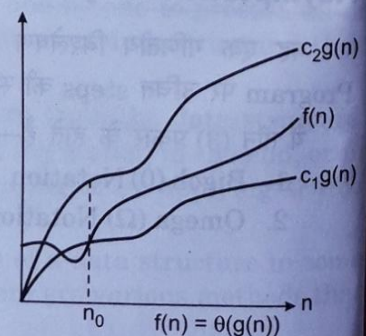
### 3. Theta (θ) Notation :

$\theta(g(n)) = f(n)$  जहाँ  $C_1$  और  $C_2$  दो constants हैं जो इस प्रकार हैं कि

$0 \leq C_1(g(n)) \leq f(n) \leq C_2(g(n))$  सभी  $n \geq n_0$  के लिए;  $f(n) \rightarrow$  एक ऐसा फलन है जो किसी Algorithm का Actual Running time है।

$g(n) \rightarrow$  एक ऐसा फलन है जो किसी Algorithm द्वारा वास्तविक समय (Running time) लिया गया।

लेकिन  $\theta$  notation में Average Case होता है मतलब Algo में कुल step ( $n$ ) के साथ  $f(n)$  का मान किसी Constant के गुणांक के साथ minimum और maximum value के बीच exist करता है।





Minimum मान के साथ Algo का running time  $C_1g(n)$

Maximum मान के साथ Algo का running time  $C_2g(n)$

$n_0 \rightarrow$  एक ऐसा स्थिर value होता है जहाँ से हमारी  $\theta$  notation की condition पूरी होती है।

$C_1$  और  $C_2$  दो अलग-अलग values हैं जो यदि Running time Algo function के साथ हों तो  $f(n)$  का मान मध्य में आएगा।

### Example :

```
void main ( )—————12
{—————1
    int a, b, c ;————— $n^2$ 
    printf ("Enter Two Values"); ———  $2n$ 
    scanf ("%d%d", &a, &b); ———  $3n$ 
    c = a + b ;—————  $n/2$ 
    printf ("The value of c is %d", c) ; —  $n + 1$ 
    getch ( ) ; —————  $n + 2$ 
} —————1
```

$$\text{function } f(n) = n + 1 + n^2 + 2n + 3n + n/2 + n + 1 + n + 2 + 1$$

$$g(n) = n^2 + n + 1$$

$n_0$  एक ऐसा boundary value है, जहाँ के बाद  $n$  का मान +ve आता है।

$C_1$  और  $C_2$  दो values हैं जो  $g(n)$  के साथ लगने पर minimum और maximum value के बीच आता है।

माना, उपरोक्त Program (Algo) को run करने का Actual time line = 10 sec.

minimum time = 7 sec.

maximum time = 11 sec है अब अगर Algo 10 या 9 पर complete होता है तब  $f(n)$   $\theta$  होगा।

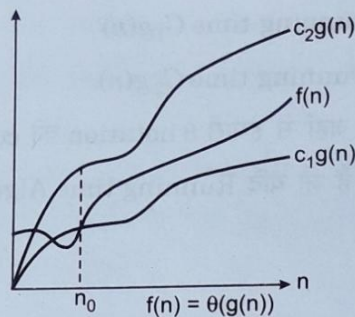
## Algorithm Performance

The performance evaluation of an algorithm is obtained by totalling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size  $n$  and it is to be considered modulo a multiplicative constant. The following notations are commonly used in performance analysis and used to characterize the complicity of an algorithm.

### $\theta$ -Notation

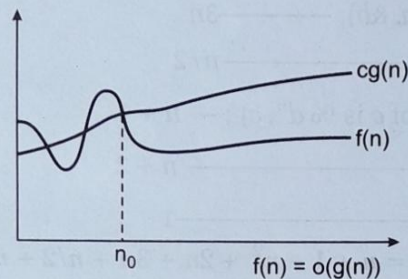
This notation bounds a function to within constant factors. We say  $f(n) = \theta(g(n))$  if there exists positive constant  $n_0$ ,  $c_1$  and  $c_2$  such that to the right of  $n_0$  the value of  $f(n)$  always lies between  $c_1g(n)$  and  $c_2g(n)$  inclusive.





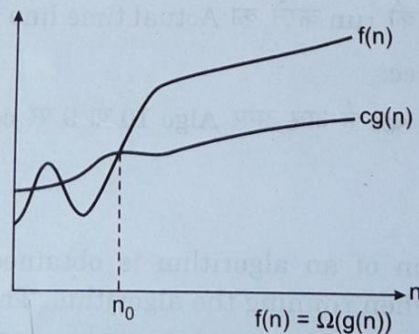
### O-Notation (Upper bound)

This notation gives up upper bound for a function to within a constant factor. We write  $f(n) = o(g(n))$  if there are positive constant  $n_0$  and  $c$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies on or below  $(g(n))$ .



### Ω-Notation (Lower bound)

This notation gives a lower bound for a function to within a constant factor. We write  $f(n) = \Omega g(n)$  if there are positive constants  $n_0$  and  $c$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies on or above  $cg(n)$ .



## Algorithmic Notation

An algorithm, is a finite step-by-step list of well-defined instructions for solving a particular problem. The formal definition of an algorithm which uses the notation of a turing machine or its equivalent.

The format for the formal presentation of an algorithm consists of two parts. The first part is a paragraph which tells the purpose of the algorithm, identifies the variables which occur in the algorithm and lists the input data. The second part of the algorithm consists of the list of steps that is to be executed.



**Comments :** Each step may contain a comment in brackets which indicates the main purpose of the step. The comment will usually appear at the beginning or at the end of the step.

**Variable names :** Variable names will use capital letters, as in MAX and DATA. Single-letter names of variables used as counters or subscripts will also be capitalized in the algorithms (K and N, for Example), even though lowercase may be used for these same variables (k & n) in the accompanying mathematical description and analysis.

**Assignment statement :** Our assignment statement will use the dots-equal notation (: =) that is used in Pascal. For example :

Max : = Data [1]

Assigns the value in Data [1] to MAX. Some texts use the backward Arrow ( $\leftarrow$ ) or the equal sign (=) for this operation.

**Input and output :** Data may be input and assigned to variables by means of a Read statement with the following form :

Read : Variable\_names

Similarly, messages, placed in quotation marks, and data in variables may be output by means of a write or print statement with the following form :

Write/print : "messages/variablename"

**Procedures :** The term "procedure" will be used for independent algorithmic module which solves a particular problem. The use of the word "Procedure" or "module" rather than "algorithm" for a given problem is simply a matter of taste.

## What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is :

type \*var\_name;

Here, type is the pointer's base type; it must be a valid C data type and var\_name is the name of the pointer variable. The asterisk \* used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. take a look at some of the valid pointer declarations—

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch; /* pointer to a character */
```

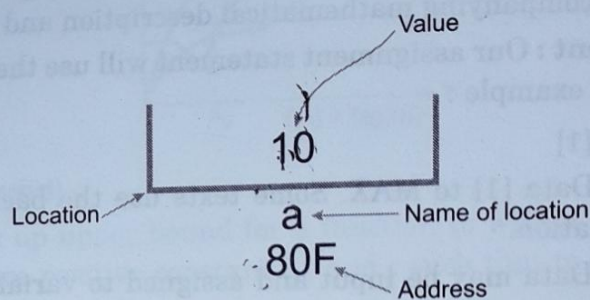
The actual data type of the value of all pointers, whether integer, float, character, or double otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.



Whenever a **variable** is declared in a program, system allocates a location i.e. an address to that variable in the memory, to hold the assigned value. This location has its own address number which we just show above.

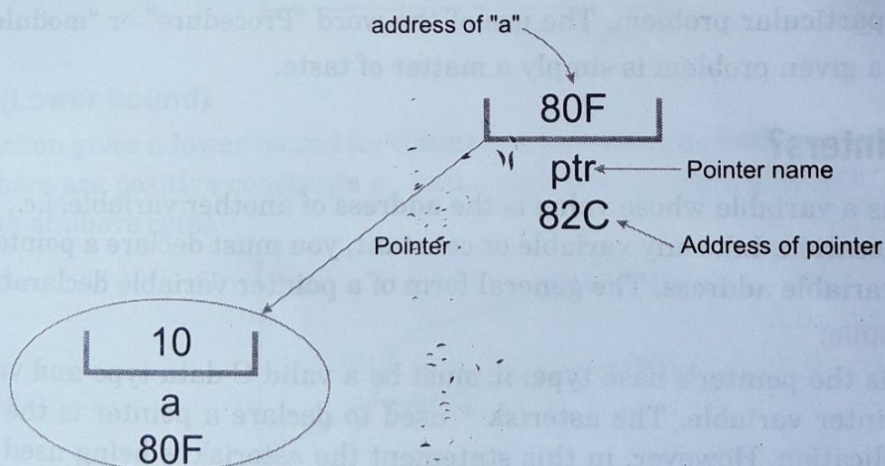
Let us assume that system has allocated memory location 80F for a variable *a*.

`int a = 10;`



We can access the value 10 either by using the variable name *a* or by using its address 80F. The question is how we can access a variable using its address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses, are called **Pointer variables**.

A pointer variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.



## How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently.

- We define a pointer variable,
- assign the address of a variable to a pointer and
- finally access the value at the address available in the pointer variable.

This is done by using unary operator `*` that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations :



```
#include <stdio.h>
void main ()
{
    int var = 20; /* actual variable declaration */
    int *ip; /* pointer variable declaration */

    ip = &var; /* store address of var in pointer variable*/

    printf ("Address of var variable: %x\n", & var);

    /* address stored in pointer variable */
    printf ("Address stored in ip variable: %x\n", ip);

    /* access the value using the pointer */
    printf ("Value of *ip variable: %d\n", *ip);
}
```

When the above code is compiled and executed, it produces the following result—

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of \*ip variable: 20

## NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an 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>
void main ()
{
    int *ptr = NULL;
    printf ("The value of ptr is : %x\n", ptr);
}
```

When the above code is compiled and executed, it produces the following result—

The value of ptr is 0.

In 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.

### Key Points to remember about pointers in C :

- ❖ Normal variable stores the value whereas pointer variable stores the address of the variable.
- ❖ The content of the C pointer always be a whole number *i.e.* address.
- ❖ The value of null pointer is 0.
- ❖ & symbol is used to get the address of the variable.
- ❖ \* symbol is used to get the value of the variable that the pointer is pointing to.
- ❖ If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- ❖ Two pointers can be subtracted to know how many elements are available between these two pointers.
- ❖ It allows C language to support Dynamic Memory Management.

### Example of Pointer demonstrating the use of & and\*

```
#include <stdio.h>
int main ( )
/* Pointer of integer type, this can hold the address of an integer type variable.
*/
int *p;
int var = 10;
/* Assigning the address of variable var to the pointer p. The p can hold the address of
var because var * is an integer type variable.
*/
p = &var;
printf ("Value of variable var is: %d", var);
printf ("\n value of variable var is: %d", *p);
printf ("\nAddress of variable var is: %p", &var);
printf ("\nAddress of variable var is: %p", p);
printf ("\nAddress of pointer p is: %p", &p);
return 0;
}
```

### Output :

Value of variable var is: 10

Value of variable var is: 10

Address of variable var is: 0x7fff5ed98c4c

Address of variable var is: 0x7fff5ed98c4c

Address of pointer p is: 0x7fff5ed98c50



## Variables in C Language

When we want to store any information (data) on our computer/laptop, we store it in the computer's memory space. Instead of remembering the complex address of that memory space where we have stored our data, our operating system provides us with an option to create folders, name them, so that it becomes easier for us to find it and access it.

Similarly, in C language, when we want to use some data value in our program, we can store it in a memory space and name the memory space so that it becomes easier to access it.

The naming of a address is known as **variable**. Variable is the name of memory location. Unlike constant, variables are changeable, we can change value of a variable during execution of a program. A programmer can choose a meaningful variable name. Example : average, height, age, total etc.

### Declaring, Defining and Initializing a variable

**Declaration** of variables must be done before they are used in the program. Declaration does the following things :

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. Until the variable is defined the compiler doesn't have to worry about allocating memory space to the variable.
4. Declaration is more like informing the compiler that there exists a variable with following datatype which is used in the program.

Syntax :

Datatype variable\_name;

**Initializing** a variable means to provide it with a value. A variable can be initialized and defined in a single statement, like:

Example :

```
int a = 10;
```

Let's write a program in which we will use some variables.

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

```
// Variable declaration (optional)
```

```
extern int a, b;
```

```
extern int c;
```

```
void main ( )
```

```
{
```

```
/* variable definition: */
```

```
int a, b;
```

```
/* actual initialization */
```

```
a = 7;
```

```
b = 14;
```

```
/* using addition operator */
```

```
c = a + b;
```



```

/* display the result*/
printf("Sum is : %d\n", c);
}

```

Sum is : 21

### Types of variable :

1. Local Variables
2. Global Variables

#### Local Variables

A variable is said to be a local variable if it is declared inside a function or inside a block. The scope of the local variable is within the function or block in which it was declared. A local variable remains in memory until the execution of the function or block in which it was declared is completed. Let's see the following example:

```

main ( )
{
    int x;
    printf("x = %d", x);
}

```

In the above example, the variable *x* is a local variable with respect to the main function. Variable *x* is not accessible outside main function and *x* remains in the memory until the execution of the main function completes.

#### Global Variables

A variable is said to be a global variable if it is declared outside all the functions in the program. A global variable can be accessed throughout the program by any function. A global variable remains in the memory until the program terminates. In a multi-file program, a global can be accessed in other files wherever the variable is declared with the storage class **extern**.

```

#include<stdio.h>
#include<conio.h>
int g = 10;
void main ( )
{
    int x = 20;
    clrscr ( );
    printf("inside main, g = %d", g);
    printf("\nInside main, x = %d", x);
    {
        int y = 30;
        printf("\nInside block, g = %d", g);
    }
}

```

```

pri
pri
}
print
/*pri
print
getch
}

```

In th  
and y is  
That is  
been co

### Rules

- 1.
- 2.
- 3.

### Const

Co  
fixed v



### Type

1



```

printf("\nInside block, y = %d", y);
printf("\nInside block, x = %d", x);
}
printf("\nOutside block, g = %d", g);
/*printf("Outside block, y = %d", y); */
printf("\nOutside block, x = %d", x);
getch ();
}

```

In the above example, **g** is a global variable and **x** is a local variable with respect to **main** and **y** is local variable within the block. The variable **y** cannot be accessed outside the block. That is why the **printf** statement outside the block accessing the value of the variable **y** has been commented out.

## Rules for Naming a Variable in C

1. A variable name can have letters (both uppercase and lowercase letters), digits and underscore only.
2. The first letter of a variable should be either a letter or an underscore. However, it is discouraged to start variable name with an underscore. It is because variable name that starts with an underscore can conflict with system name and may cause error.
3. There is no rule on how long a variable can be. However, only the first 31 characters of a variable are checked by the compiler. So, the first 31 letters of two variables in a program should be different.

## Constant

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

- ❖ C Constants are also like normal variables. But, only difference is, their values cannot be modified by the program once they are defined.
- ❖ Constants refer to fixed values. They are also called as literals
- ❖ Constants may be belonging to any of the data type.
- ❖ Syntax:

```

const data_type variable_name;
(or)
const data_type *variable_name;

```

## Types of C Constant

1. Integer constants
2. Character constants
3. String constants
4. Symbolic constants

**1. Integer Constants :** There are **two** types of numeric constants,

1. Integer constants
2. Real or floating-point constants



## Integer constants

- ❖ Any whole number value is an integer.
- ❖ An integer constant refers to a sequence of digits without a decimal point.
- ❖ An integer preceded by a unary minus may be considered to represent a negative constant.

**Example :** 0      -33      32767

There are three types of integer constants namely,

- (a) Decimal integer constant
- (b) Octal integer constant
- (c) Hexadecimal integer constant

### Decimal Integer constant (base 10)

It consists of any combinations of digits taken from the set 0 through 9, preceded by an optional - or + sign.

The first digit must be other than 0.

Embedded spaces, commas, and non-digit characters are not permitted between digits.

**Valid :**      0      32767      -9999      -23

**Invalid :**

- 12,245      —      Illegal character (,)
- 10 20 30      —      Illegal character (blank space)
- 045      —      First digit cannot be zero.

### Octal Integer Constant (base 8)

- ❖ It consists of any combinations of digits taken from the set 0 through 7.
- ❖ If a constant contains two or more digits, the first digit must be 0.
- ❖ In programming, octal numbers are used.

**Valid :**      037      0      0435

**Invalid :**

- 0786      -      Illegal digit 8
- 123      -      Does not begin with zero
- 01.2      -      Illegal character (.)

### Hexadecimal integer constant (base 16)

- ❖ It consists of any combinations of digits taken from the set 0 through 7 and also a through f (either **uppercase** or **lowercase**).
- ❖ The letters a through f (or A through F) represent the decimal quantities 10 through 15 respectively.
- ❖ This constant must begin with either 0x or 0X.
- ❖ In programming, hexadecimal numbers are used.

**Valid Hexadecimal Integer Constant:**      0x      0X1      0x7F

**Invalid Hexadecimal Integer constant:**

- 0xefg      -      Illegal character g
- 123      -      Does not begin with 0x



**Unsigned integer constant :** An unsigned integer constant specifies only positive integer value. It is used only to count things. This constant can be identified by appending the letter u or U to the end of the constant.

Valid:      **0u**            **1U**            **65535u**            **0x233AU**

Invalid:    **-123**        **-**            **Only positive value**

#### Example of integer :

```
#include<stdio.h>
#include<conio.h>
int main ()
{
const int a = 1234;
clrscr ();
printf ("Value of a = %d", a);
getch();
return 0;
}
```

## 2. Character Constant :

A character constant is a single character, enclosed in single quotation marks.

e.g., 'A' 'B' '1'

Characters are stored internally in computer as coded set of binary digits, which have positive decimal integer equivalents. The value of a character constant is the numeric value of the character in the machine's character set. This means that the value of a character constants can vary from one machine to the next, depending on the character set being used on the particular machine. For example, on ASCII machine the value of 'A' is 65 and on EBCDIC machine it is 193.

#### Example of Character

```
#include<stdio.h>
#include<conio.h>
void main()
{
const char a = 'X';
clrscr();
printf("Value of a = %c, d");
getch();
}
```

#### Output :

Value of a = X

## 3. String constant :

- ❖ A character string, a string constant consists of a sequence of characters enclosed in double quotes.



- ❖ A string constant may consist of any combination of digits, letters, escaped sequences and spaces. Note that a character constant 'A' and the corresponding single character string constant "A" are not equivalent.

'A' - Character constant - 'A'  
 "A" - String Constant - 'A' and '\0' (NULL)

The string constant "A" consists of character A and \0. However, a single character string constant does not have an equivalent integer value. It occupies two bytes, one for the ASCII code of A and another for the NULL character with a value 0, which is used to terminate all strings.

Valid String constants :	"W"	"100"	"24, Kaja Street"
Invalid String constants :	"W		the closing double quotes missing
	Raja"		the beginning double quotes missing

### Rules for Constructing String Constants

1. A string constant may consist of any combination of digits, letters, escaped sequences and spaces enclosed in double quotes.
2. Every string constant ends up with a **NULL** character which is automatically assigned (before the closing double quotation mark) by the compiler.

### Symbolic Constant :

A symbolic constant is name that substitute for a sequence of character that cannot be changed. The character may represent a numeric constant, a character constant, or a string. When the program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. They are usually defined at the beginning of the program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc., that the symbolic constants represent.

For example,

a **C program** consists of the following symbolic constant definitions.

```
#define PI 3.141593
```

```
#define TRUE 1
```

```
#define FALSE 0
```

#define PI 3.141593 defines a symbolic constant PI whose value is 3.141593. When the program is preprocessed, all occurrences of the symbolic constant PI and replaced with the replacement text 3.141593.

Note that the **preprocessor statements** begin with a #symbol, and are not end with a semicolon. By convention, **preprocessor** constants are written in UPPERCASE.

#### Examples

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

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

```
#define TRUE 1
```

```
#define PI 3.141593
```



```

void main ()
{
float  a;
float  b;
float  c;
float  d = Pl;
clrscr();
if (TRUE)
{
a = 100;
b = a*10;
c = b - a;
}
printf("\na = %f\nb=%f\nc=%f\nPl=%f", a, b, c, d);
getch();
}

```

## Arrays

When there is a need to use many variables, there is a big problem because we will conflict with name of variables so that in this situation where we want to operate on many numbers then we can use array. The number of variables also increases the complexity of the Program so that we use Arrays.

Arrays are set of elements having same data type or we can say that Arrays are collection of elements having same name and same data type. But always remember Arrays always start from its index value and the index of array starts from 0 to  $n-1$ .

Suppose we want to access 5th element of array then we will use 4th element because Arrays start from 0 and arrays are always stored in Continuous Memory Locations. The number of elements and types of array are identified by subscript of array elements.

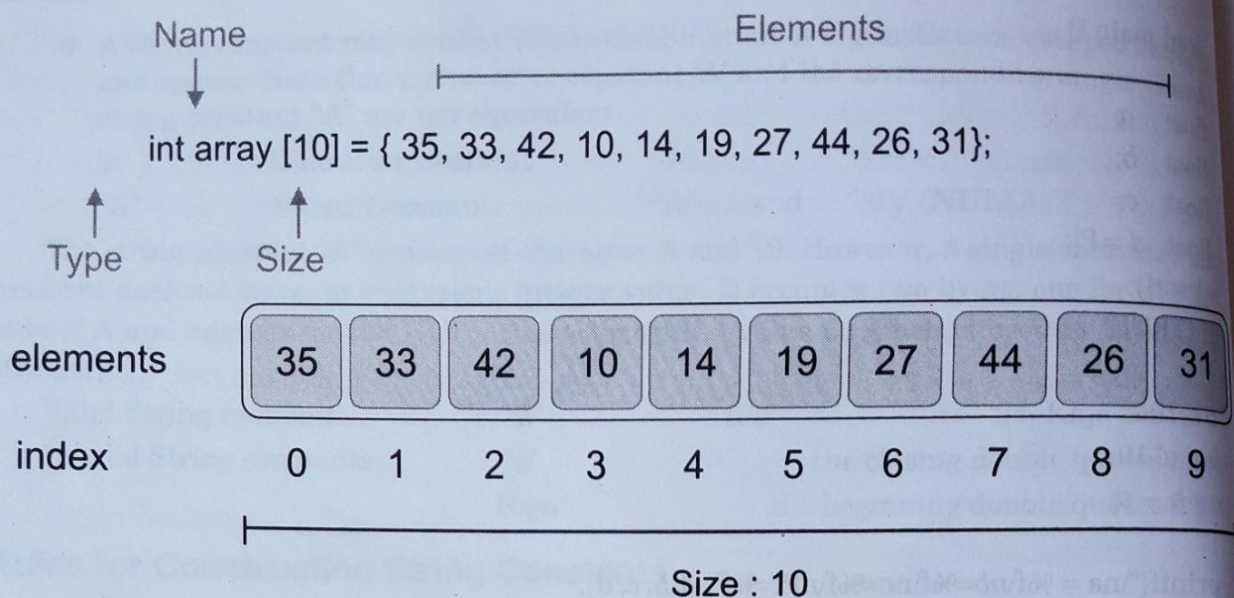
**Array is a container which can hold a fixed number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array :**

- ❖ **Element :** Each item stored in an array is called an element.
- ❖ **Index :** Each location of an element in an array has a numerical index, which is used to identify the element.

### Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.





As per the above illustration, following are the important points to be considered :

- ❖ Index starts with 0.
- ❖ Array length is 10 which means it can store 10 elements.
- ❖ Each element can be accessed via its index. For example, we can fetch an element at index 0 as 9.

#### How to declare an array in C programming :

The general form of declaring a simple (one dimensional) array is.....

`+array_type variable_name [ array_size];`

in your C /C ++ program you can declare an array like

`int Age [10];`

Here `array_type` declares base type of array which is the type of each element in array. In our example `array_type` is `int` and its name is `Age`. Size of the array is defined by `array_size` i.e. 10.

We can access array elements by index, and first item in array is at index 0. First element of array is called lower bound and its always 0. Highest element in array is called upper bound.

In C programming language upper and lower bounds cannot be changed during the execution of the program, so array length can be set only when the program is written.

Age 0	Age 1	Age 2	Age 3	Age 4	Age 5	Age 6	Age 7	Age 8	Age 9
30	32	54	32	26	29	23	43	34	5

Array has 10 elements

**Note :** One good practice is to declare array length as a constant identifier. This will minimize the required work to change the array size during program development.



Considering the array we declared above we can declare it like

1. # define NUM \_ EMPLOYEE 10
2. int Age [NUM \_ EMPLOYEE];

How to initialize an array in C program :

Initialization of array is very simple in C programming. There are two ways you can initialize arrays.

- ❖ Declare and initialize array in one statement.
- ❖ Declare and initialize array separately.

Look at the following C code which demonstrates the declaration and initialization of an array.

1. int Age [5] = {30, 22, 33, 44, 25};
2. int Age [5];
3. Age [0] = 30;
4. Age [1] = 22;
5. Age [2] = 33;
6. Age [3] = 44;
7. Age [4] = 25;

Array can also be initialized in a way that array size is omitted, in such case compiler automatically allocates memory to array.

```
int Age[ ]= {30, 22, 33, 44, 25};
```

Let's write a simple program that uses arrays to print out number of employees having salary more than 3000.

**The various types of Array are provided by C as follows :**

1. Single Dimensional Array
2. Two Dimensional Array
3. Three Dimensional array

### 1. Single Dimensional Array

A dimensional array is used for representing the elements of the array. For example,

```
int a[5];
```

The [] is used for dimensional or the sub-script of the array that is generally used for declaring the elements of the array. For accessing the element from the array we can use the subscript of the Array like this

```
a [3]= 100;
```

This will set the value of 4<sup>th</sup> element of array

So there is only the single bracket, then it is called the Single Dimensional Array.

This is also called as the Single Dimensional Array

**Syntax :** data-type arr\_name[array\_size];



Array declaration, initialization and accessing	Example
<p><b>Array declaration syntax :</b>  data_type arr_name [arr_size];</p> <p><b>Array initialization syntax :</b>  data_type arr_name [arr_size]=(value1, value2, value3,...);</p> <p><b>Array accessing syntax :</b>  arr_name[index];</p>	<p><b>Integer array example:</b>  int age [5];  int age [5] = {0,1,2,3,4};  age [0]; /*0 is accessed */  age [1]; /*1 is accessed */  age [2]; /*2 is accessed */</p> <p><b>Character array example :</b>  char str [10];  char str [10]= {'H', 'a', 'i'};  (or)  char str [0] = 'H';  char str[1] = 'a';  char str[2] = 'i';</p> <p>str [0]; /*H is accessed */  str [1]; /*a is accessed */  str [2]; /*i is accessed */</p>

#### EXAMPLE PROGRAM FOR ONE DIMENSIONAL ARRAY IN C :

```
#include<stdio.h>
void main()
{
    int i;
    int arr [5] = {10, 20, 30, 40, 50};
        //declaring and initializing array in C
        //To initialize all array elements to 0, use int arr [5] = {0};
        /* Above array can be initialized as below also
arr[0] = 10;
arr [1] = 20;
arr [2] = 30;
arr [3] = 40;
arr [4] = 50;*/
for (i=0;i<5; i++)
{
    //Accessing each variable
    printf ("value of arr[%d] is %d\n", i, arr[i]);
}
}
```



**Output :**

Value of arr [0] is 10  
 value of arr [1] is 20  
 value of arr [2] is 30  
 value of arr [3] is 40  
 value of arr [4] is 50

**2. Two Dimensional Array or the Matrix**

The Two Dimensional array is used for representing the elements of the array in the form of the rows and columns and these are used for representing the Matrix A. Two Dimensional Array uses the two subscripts for declaring the elements of the Array

Like this int a [3] [3]

So this is the example of the Two Dimensional Array. In this first 3 represents the total number of Rows and the second elements represents the total number of Columns. The total number of elements are judged by multiplying the number of Rows \* Number of Columns in the Array in the above array. The total number of elements is 9.

❖ syntax : data-type array\_name[num\_of\_rows][num\_of\_columns];

Array declaration, initialization and accessing	Example
<b>Array declaration syntax :</b> data_type arr_name [num_of_rows] [num_of_columns]; <b>Array initialization syntax :</b> data_type arr_name[2][2] = {{0,0}, {0,1}, {1,0}, {1,1}}; <b>Array accessing syntax :</b> arr_name[index];	<b>Integer array example :</b> int arr[2][2]; int arr [2][2]={{1, 2}, {3, 4}};  arr [0] [0]=1; arr[0] [1] =2; arr [1] [0]=3; arr [1] [1] =4;

**EXAMPLE PROGRAM FOR TWO DIMENSIONAL ARRAY IN C :**

```
#include<stdio.h>
void main()
{
    int i,j;
    //declaring and Initializing array
    int arr [2][2] = {10, 20, 30, 40};
    /* Above array can be initialized as below also
    arr [0][0]=10;//Initializing array
    arr [0] [1] =20;
    arr [1] [0] =30;
    arr [1] [1] =40;*/
    for (i=0;i<2;i++)
    {
```



```

for (j=0;j<2;j++)
{
    //Accessing variables
    printf ("value of arr [%d] [%d] : %d\n", i,j,arr [i] [j]);
}
}

```

**Output :**

value of arr [0] [0] is 10  
 value of arr [0] [1] is 20  
 value of arr [1] [0] is 30  
 value of arr [1] [1] is 40

**3. Multidimensional or the Three Dimensional Array**

The Multidimensional Arrays are used for representing the total number of Tables or Matrix-A. Three dimensional Array is used when we want to make the two or more tables or the Matrix elements for declaring the Array elements we can use the way like this

```
int a [3] [3] [3];
```

In this first 3 represents the total number of Tables and the second 3 represents the total number of rows in each table and the third 3 represents the total number of columns in the Tables.

So this makes the 3 Tables having the three rows and the three columns.

The main and very important thing about the array that the elements are stored always in the contiguous in the memory of the computer

A multidimensional array is declared using the following syntax :

**type array\_name [d1] [d2] [d3] [d4] ....[dn];**

where each **d** is a dimension, and **dn** is the size of final dimension.

**Examples :**

1. **int table [5] [5] [20];**
2. **float arr [5] [6] [5] [6] [5];**

**In Example 1 :**

- ❖ **int** designates the array type integer.
- ❖ **table** is the name of our 3D array.
- ❖ Our array can hold 500 integer-type elements. This number is reached by multiplying the value of each dimension. In this case :  $5 \times 5 \times 20 = 500$ .

**In Example 2 :**

- ❖ Array **arr** is a five-dimensional array.
- ❖ It can hold 4500 floating-point elements ( $5 \times 6 \times 5 \times 6 \times 5 = 4500$ ).



## Initializing a 3D Array in C

Like any other variable or array, a 3D array can be initialized at the time of compilation. By default, in C, an uninitialized 3D array contains “garbage” values, not valid for the intended use.

Let's see a complete example on how to initialize a 3D array :

### Syntax :

```
#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:::\n\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();
}

```

Print :

```

Turbo C++ IDE
:::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

```

### Storing Values in a Continuous Location Using a Loop

The pointer syntax above assigns values to a particular location of an array, but if you want to store values in multiple locations automatically then you should use a loop.

Here is an example using the **for loop** command:

?

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int i,j, k,x=1;
    int arr [3] [3] [3] ;
    clrscr ();
    printf (":::3D Array Elements:::\n\n");
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            for (k=0;k<3;k++)
            {

```



```

arr[i][j][k]=x;
printf("%d\t",arr [i] [j] [k];
x++;
}
printf("\n");
}
printf("\n");
}
getch();
}

```

### Basic Operations :

Following operations can be performed on arrays :

1. Traversing
2. Searching
3. Insertion
4. Deletion
5. Sorting
6. Merging

**1. Traversing :** It is used to access each data item exactly once so that it can be processed.

E.g.

We have linear array A as below :

1	2	3	4	5
10	20	30	40	50

Here we will start from beginning and will go till last element and during this process we will access value of each element exactly once as below :

A[1] = 10

A[2] = 20

A[3] = 30

A[4] = 40

A[5] = 50

**2. Searching :** It is used to find out the location of the data item if it exists in the given collection of data items.

E.g.

We have linear array A as below :

1	2	3	4	5
15	50	35	20	25

Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here :

1. Compare 20 with 15

20 != 15, go to next element.



### 38 ■ Data Structure Using C

2. Compare 20 with 50  
20  $\neq$  50, go to next element.
3. Compare 20 with 35  
20  $\neq$  35, go to next element.
4. Compare 20 with 20  
20 == 20, so 20 is found and its location is 4.

**3. Insertion :** It is used to add a new data item in the given collection of data items.

E.g.

We have linear array A as below :

1	2	3	4	5
10	20	50	30	15

New element to be inserted is 100 and location for insertion is 3. So shift the elements from 5th location to 3rd location downwards by 1 place. And then insert 100 at 3rd location. It is shown below :

1	2	3	4	5	6
10	20	50	30	15	15

1	2	3	4	5	6
10	20	50	30	30	15

1	2	3	4	5	6
10	20	50	50	30	15

1	2	3	4	5	6
10	20	100	50	30	15

↑

**4. Deletion :** It is used to delete an existing data item from the given collection of data items.

E.g.

We have linear array A as below :

1	2	3	4	5	6
10	20	50	40	25	60

The element to be deleted is 50 which is at 3rd location. So shift the elements from 4th to 6th location upwards by 1 place. It is shown below :



1	2	3	4	5	6
10	20	40	40	25	60

1	2	3	4	5	6
10	20	40	25	25	60

1	2	3	4	5	6
10	20	40	25	60	60

After deletion the array will be :

1	2	3	4	5	6
10	20	40	25	60	

**5. Sorting :** It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

E. g.

We have linear array A as below :

1	2	3	4	5
10	50	40	20	30

After arranging the elements in increasing order by using a sorting technique, the array will be :

1	2	3	4	5
10	20	30	40	50

**6. Merging :** It is used to combine the data items of two sorted files into single file in the sorted form.

We have sorted linear array A as below :

1	2	3	4	5	6
10	40	50	80	95	100

And sorted linear array B as below :

1	2	3	4
20	35	45	90

After merging merged array C is as below :

1	2	3	4	5	6	7	8	9	10
10	20	35	40	45	50	80	90	95	100



## Insertion Operation in Array

Insertion operation is used to insert a new element at specific position into one dimensional array.

In order to insert a new element into one dimensional array we have to create space for new element.

Suppose there are N elements in an array and we want to insert a new element between first and second elements. We have to move last N-1 elements down in order to create space for the new element.

### Algorithm For Insertion Operation in Array

Step 1 :  $TEMP = N - 1$

$POS = POS - 1$

Step 2 : Repeat Step 3 while  $TEMP \geq POS$

Step 3 :  $A[TEMP + 1] = A[TEMP]$

$TEMP = TEMP - 1$

Step 4 :  $A[POS] = X$

Step 5 :  $N = N + 1$

### Program For Insertion Operation in Array

```
#include <stdio.h>
#include <conio.h>
#define N 5
void main()
{
    int a[N] = {10, 20, 30, 40, 50};
    int POS, x;
    void traverse (int*a, int n);
    void insert (int*a, int POS, int x);
    clrscr ();
    printf("Before Insertion\n");
    traverse (a, N);
    printf("Enter Position:");
    scanf("%d", &POS);
    printf("Enter Value:");
    scanf("%d", &x);
    insert(a, POS, x);
    printf("After Insertion\n");
    traverse (a, N+1);
    getch();
}

void insert (int*a, int POS, int x)
```



```

{
    POS=POS-1;
    int TEMP =N-1;
    while (TEMP>=POS)
    {
        a [TEMP+1]=a [TEMP];
        TEMP=TEMP-1;
    }
    a[POS]=x;
}

void traverse (int *a, int n)
{
    int START =0;
    while (START<n)
    {
        printf("%d\n",a[START]);
        START=START+1;
    }
}

```

**Algorithms for array traversal, insertion and deletion are shown below separately**

**ALGORITHM : (Traversing of Linear Array)** Here Arr is a linear array with lower bound L and upper bound U :

1. [Initialize Counter] Set  $K=L$
2. Repeat steps 3 and 4 while  $K \leq U$
3. Print Arr [K].
4. [Increase Counter] Set  $K=K+1$ .
5. Exit

**ALGORITHM : (Inserting into a Linear Array) INSERT (Arr, N, K, ITEM)**

Here Arr is a linear array with N elements, K is positive integer such that  $K \leq N$  and ITEM is the element to be inserted.

1. [Initialize Counter] Set  $I=N$ .
2. Repeat steps 3 and 4 while  $I \geq K$ .
3. Set Arr [I+1]=Arr [I].
4. [Decrease Counter] Set  $I=I-1$ .
5. [Inserting ITEM] Set Arr [K] =ITEM.
6. [Reset Number of Elements]. Set  $N=N+1$ .
7. Exit

**ALGORITHM : (Deleting From Linear Array) DELETE (Arr, N, K)**



Here Arr is a linear array with N elements, K is positive integer such that  $K \leq N$  and Kth element is to be deleted.

1. [Initialize Counter] Set  $I=K$ .
2. Repeat steps 3 and 4 while  $I=N-1$ .
3.  $Arr[I]=Arr[I+1]$ .
4. [Increase Counter] Set  $I=I+1$ .
5. [Reset Number of Elements]. Set  $N=N+1$ .
6. Exit.

## TRAVERSAL

Traversal means visiting each element of array exactly once. Processing of array like printing the contents or counting the no. of elements in array.

### Algorithm

1. [Initialize Counter] Set  $k=LB$ .
2. Repeat steps 3 and 4 while  $k \leq UB$
3. [visit the element] apply process to a  $[k]$
4. [increase Counter] Set  $k=k+1$ .
5. [end of step 2 loop]
6. exit.

### Alternative Algorithm

1. Repeat for  $k=LB$  to  $UB$
2. Apply process to  $A[k]$
3. [end of loop]
4. exit

## INSERTION

For inserting an element in 1-D array, one has to shift elements one position down.

### Algorithm

1.  $temp = number$ . // number is the no. of elements already present in array.
2. [scan the list for the position of new element]
3. repeat while  $temp > position$  // actual in array
4. [move the last element one position down]
  - ❖  $array[temp] = array[temp-1]$
  - ❖  $temp=temp-1$
5. [insert the element]
6.  $array[position]=no$
7. [change no. of informations by 1]
8.  $number = number + 1$
9. return



## DELETION

For deleting element in 1-D array the elements have to be shifted position up.

### Algorithm

1. [initialization]
2. element=array [position//keep information of deleted element]
3. [make copy of position]
4. temp=position
5. [update the list, number represents no. of items in list]
6. repeat while temp<number-1 //number is no. of elements in array
  - ❖ array [temp]=array[temp+1]
  - ❖ temp=temp+1;
7. [decrease the size of info list by one]
8. number=number-1;
9. exit

## SEARCHING

Searching means finding an element in an array. It is of two kinds :

- ❖ Linear search
- ❖ binary search

### Linear Search

1. repeat steps 2 and 3 while loc<number
2. if array[loc]=item [search for element in array]
3. print "element found at position loc"and exit  
[end of if structure]
4. set loc=loc +1  
[end of step 1 loop]
5. if loc>=number
6. display"element not found"
7. exit

### Binary Search in One dimensional array

In binary search we take a sorted array. In binary search we split the array into half and compare with the middle element.

### Algorithm

1. set beg =LB, end=UB and mid=int ((beg+end)/2)
2. repeat steps 3 and 4 while beg <=end and a [mid]!=item.
3. if item <=a [mid] then  
set end =mid-1  
else



- ```

    set beg = mid-1
    [end of if structure]
4. set mid =int ((beg+end)/2)
   [end of step 2 loop]
5. if a [mid] =item then
    set loc=mid
    [end of if structure]
6. exit

```

### **SORTING-Algorithm for Bubble Sort**

1. repeat steps 2 and 3 for k=1 to n-1
2. set ptr=1 [initialize pass pointer ptr]
3. repeat while ptr <= n-k [executes pass]
  - if data [ptr] > data [ptr +1]
  - interchange data [ptr] and data [ptr+1]
  - b) set ptr= ptr+1
  - [end of inner loop]
  - [end of step 1 outer loop]
4. Exit

### **Applications of Arrays in C**

In C programming language, arrays are used in wide range of applications. Few of them are as follows :

1. Arrays are used to store List of values.
2. In C programming language, single dimensional arrays are used to store list of values of same data type. In other words, single dimensional arrays are used to store a row of values. In single dimensional array data is stored in linear form.
3. Arrays are used to perform Matrix Operations.
4. We use two dimensional arrays to create matrix. We can perform various operations on matrices using two dimensional arrays.
5. Arrays are used to implement Search Algorithms.
6. We use single dimensional arrays to implement search algorithms like :
  1. Linear Search
  2. Binary Search
7. Arrays are used to implement Sorting Algorithms.
8. We use single dimensional arrays to implement sorting algorithms like :
  1. Insertion Sort
  2. Bubble Sort
  3. Selection Sort
  4. Quick Sort
  5. Merge Sort, etc.
9. Arrays are used to implement Data structures.
10. We use single dimensional arrays to implement data structures like :
  1. Stack Using Arrays
  2. Queue Using Arrays
11. Arrays are also used to implement CPU Scheduling Algorithms.



```
# include <stdio.h>
# include <stdlib.h>
int a [20], b [20], c [40];
int m, n, p, val, i, j, key, pos, temp;
/*Function Prototype*/
void create();
void display();
void insert();
void del();
void search();
void merge();
void sort();
int main()
{
    int choice;
    do
    {
        printf("\n\n.....Menu.....\n");
        printf("1. Create\n");
        printf("2. Display\n");
        printf("3. Insert\n");
        printf("4. Delete\n");
        printf("5. Search\n");
        printf("6. Sort\n");
        printf("7. Merge\n");
        printf("8. Exit\n");
        printf(".....");
        printf("\n Enter your choice:\t");
        scanf("%d", & choice);
        switch (choice)
        {
            case 1 :   create();
                      break;

            case 2 :
                      display();
                      break;

            case 3:
                      insert ();
                      break;

            case 4:
                      del();
```



```

                                break;

        case 5:
                                search();
                                break;

        case 6:
                                sort ();
                                break ;

        case 7:
                                merge();
                                break;

        case 8:
                                exit(0);
                                break;

        default:
                                printf("\nInvalid choice:\n");
                                break;
    }

    }while (choice!=8);

return 0;
}

void create()//creating an array
{
    printf("\nEnter the size of the array elements :\t");
    scanf("%d",&n);
    printf("\nEnter the elements for the array:\n");
    for (i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
}

//end of create()

void display()//displaying an array elements
{
    int i;
    printf("\n The array elements are : \n");
    for (i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}

//end of display()

void insert ()//inserting an element into an array

```



```

{
    printf("\nEnter the position for the new element:\t");
    scanf ("%d",&pos);
    printf("\n Enter the element to be inserted :\t");
    scanf ("%d",&val);
    for (i=n-1;i>=pos;i--)
    {
        a[i+1]=a[i];
    }
    a[pos]=val;
    n=n+1;
} //end of insert()

void del() //deleting an array element
{
    printf("\n Enter the position of the element to be deleted:\t");
    scanf ("%d",& pos);
    val=a[pos];
    for(i=pos;i<n-1;i++)
    {
        a[i]=a[i+1];
    }
    n=n-1;
    printf("\n The deleted element is =%d",val);
} //end of delete()

void search() //searching an array element
{
    printf("\n Enter the element to be searched : \t");
    scanf ("%d",& key);
    for (i=0;i<n;i++)
    {
        if (a[i] == key)
        {
            printf("\n The element is present at position %d",i);
            break;
        }
    }
    if (i==n)
    {
        printf("\n The search is unsuccessful");
    }
} //end of search()

```



```
void sort() //sorting the array elements
```

```
{
    for(i=0;i<n-1;i++)
    {
        for (j=0;j<n-1-i;j++)
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }
    }
}
```

```
printf("\n After sorting the array elements are :\n");
```

```
display ();
```

```
//end of sort
```

```
void merge() //merging two arrays
```

```
{
    printf("\n Enter the size of the second array : \t");
    scanf("%d",&m);
    printf("\n Enter the elements for the second array : \n");
    for (i=0;i<m;i++)
    {
        scanf("%d",&b[i]);
    }
    for (i=0,j=0;i<n;i++,j++)
    {
        c[j]=a[i];
    }
    for (i=0;i<m;i++,j++)
    {
        c[j]=b[i];
    }
}
```

```
p=n+m;
```

```
printf("\n Array elements after merging:\n");
```

```
for (i=0;i<p;i++)
```

```
{
    printf("%d\t",c[i]);
}
```

```
//end of merge()
```



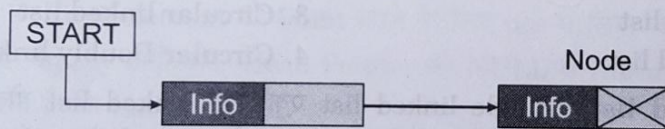
**प्रश्न 1. Linked list क्या होती है? इस पर कौन-कौन operation perform किये जाते हैं और ये कितने प्रकार की होती हैं?**

**उत्तर—**यदि किसी Program को Execution के पहले Memory को divide किया जाता है तो वह static हो जाता है और इसे change नहीं किया जा सकता, एक विशेष Data Structure जिसे linked list कहते हैं वह flexible storage system प्रदान करता है जिसको Array के प्रयोग की आवश्यकता नहीं होती है। Stack और Queue को Computer Memory में represent करने के लिए हम Array का प्रयोग करते हैं जिससे Memory को पहले से declare करना पड़ता है। इससे Memory का Wastage होता है। Memory एक महत्वपूर्ण Resource है अतः कम-से-कम use में लेकर अधिक-से-अधिक Program को run करना चाहिए।

Linked list एक विशेष प्रकार के Data elements की list होती है जो एक-दूसरे से जुड़ी होती हैं। Logical ordering हर Element को अगले element से Point करते हुए represent करते हैं जिसके दो भाग होते हैं—

1. Data Field/Information Field

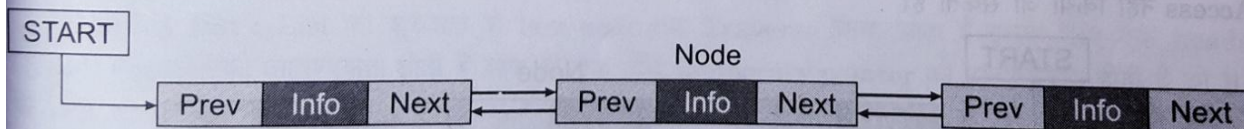
2. Link/Address/Pointer field.



यह Singly linked list है जहाँ पर एक node के दो Parts होते हैं—

1. Information/Data field

2. Link/Pointer Field



जबकि Doubly linked list में दो Pointer fields (Prev, Next) तथा एक Information field होता है।

## Operation on linked list

Linked list में Basic operation को perform किया जाता है—

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Concatenation
6. Display

**1. Creation :** यह operation linked list को बनाने के लिए use में लेते हैं। जब भी आवश्यकता हो किसी भी node को list से जोड़ दिया जाता है।



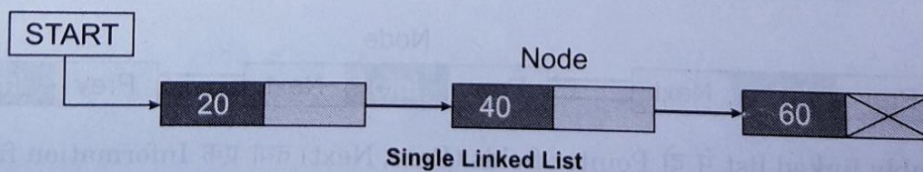
- 2. Insertion :** यह operation linked list में एक नये (New) node को विशेष स्थिति में विशेष स्थान जोड़ने के लिए काम में लिया जाता है। इस नये node को निम्न स्थानों पर Insert किया जा सकता है—
- (a) list के प्रारम्भ में
  - (b) list के अन्त में
  - (c) किसी list के मध्य (बीच) में
- 3. Deletion operation :** यह operation linked list में से किसी node को delete करने के लिए काम में लिया जाता है। Node को जिन तरीकों से delete किया जाता है वह निम्न है—
- (a) list के प्रारम्भ से।
  - (b) list के अन्त से।
  - (c) किसी list के मध्य (बीच) में
- 4. Traversing :** यह प्रक्रिया किसी linked list को एक Point से दूसरे Point तक value को print करवाने के काम आती है। अगर पहले node से आखिरी node की ओर Traversing करते हैं तो यह Forward Traversing कहलाती है।
- 5. Concatenation :** यह प्रक्रिया दूसरे list से node को जोड़ने के काम आती है। यदि दूसरी list पर node होता है तो concatenation node में  $m + n$  node होगा।
- 6. Display :** यह node प्रत्येक node की सूचना को Point करने के प्रयोग में आता है। अतः Information Part को print करना ही Display operation कहलाता है।

### Types of linked list

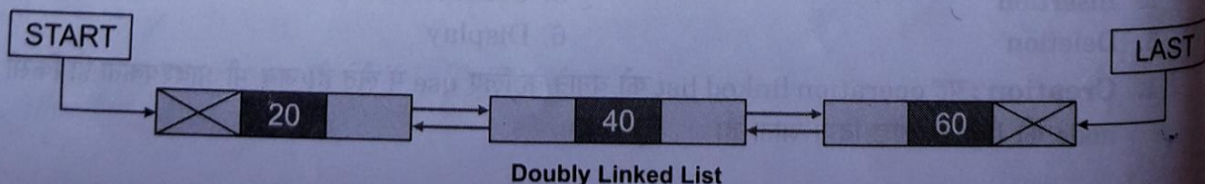
Linked list मुख्य रूप से चार प्रकार की होती हैं—

1. Single linked list
2. Doubly linked list
3. Circular linked list
4. Circular Doubly linked list

**1. Single linked list :** Single linked list एक ऐसी linked list होती है जिसमें सारे node एक-दूसरे के साथ क्रम में जुड़े होते हैं इसलिए इसे linear linked list भी कहते हैं। इसमें एक समस्या यह है कि list के आगे वाले node तो Access किया जा सकता है परन्तु एक बार Pointer के आगे जाने के बाद पीछे वाले node का Access नहीं किया जा सकता है।

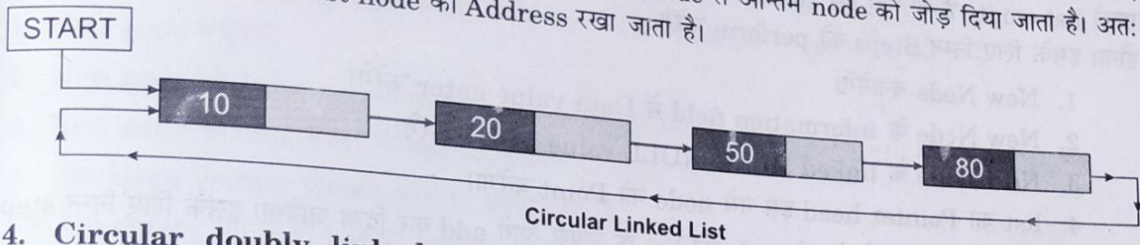


**2. Doubly linked list :** Doubly linked list वह linked list होती है जिससे सारे node एक-दूसरे के साथ multiple link के द्वारा जुड़े होते हैं। इसके अन्तर्गत list के आगे वाले node व list के पीछे वाले node दोनों को Access किया जा सकता है। इसलिए Doubly linked list में पहले node का left और अगले node के right का Address रखता है।

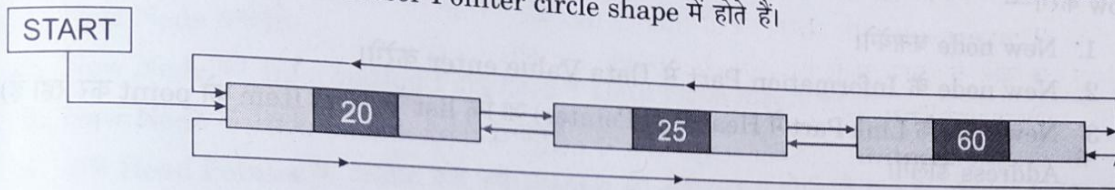




**3. Circular linked list :** Circular linked list एक ऐसी linked list होती है जिसकी कोई शुरुआत व अन्त नहीं होता है। एक Single linked list के पहले node से अन्तिम node को जोड़ दिया जाता है। अतः list के last node में list के first node का Address रखा जाता है।



**4. Circular doubly linked list :** यह एक ऐसी list होती है जिसमें दोनों Pointers (1) Successor Pointer (2) Pre-decessor Pointer circle shape में होते हैं।



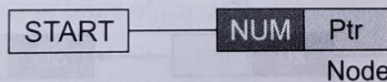
**प्रश्न 2. Single linked list पर कौन-कौन से Operations perform किये जाते हैं?**

उत्तर—

### Operations on linked list

**Create an Empty linked list :**

- I. एक node Pointer head बनाया जायेगा जिसमें किसी value को initialize नहीं किया गया है। यह Pointer list के First element को Point करने के लिए use में लिया जाता है।
- II. शुरुआत में list Empty होगी अतः Head Pointer को NULL से Initialize किया जाएगा।
- III. यह प्रदर्शित करता है कि list empty है।



**Display list :** List को शुरुआत से last node तक Traverse किया जाता है इसके लिए एक header (Head) Pointer की आवश्यकता होती है साथ ही एक और temporary pointer की आवश्यकता होती है जो list के last node तक जाता है। List का last node वह node है जिसके link part में NULL होता है अतः दो struct node pointer की आवश्यकता होगी जिसमें पहला list के First node को Point करेगा तथा दूसरा एक के बाद एक Next Node पर जाएगा और Information Part को print करेगा।

### Insertion in linked list

किसी भी element को list में insert करने के लिए सबसे पहले Free node की आवश्यकता पड़ती है। जब Free node बना लिया जाता है तो node के Information Field में Data Value को add कर देते हैं और इस नये node को इसकी जगह पर Add कर दिया जाता है। किसी भी list में Insertion 3 प्रकार से किया जा सकता है—

1. List के Beginning में
2. List के End में

3. किसी दिये गये element के बाद या पहले।

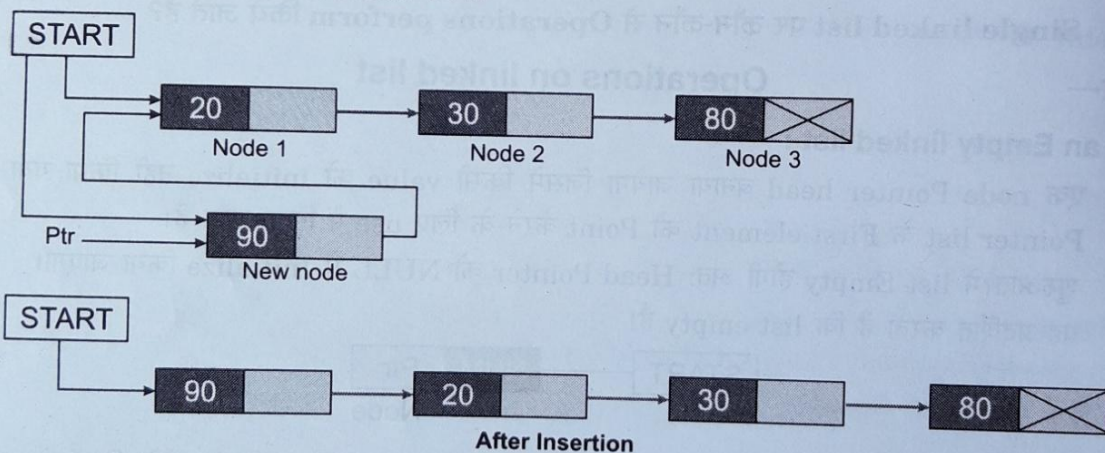


**1. Insertion at the Beginning :** किसी भी element को list के first में Add करने के लिए सबसे पहले list empty है या नहीं इसे check किया जाता है। यदि list Empty है तो नया node ही list का First node होगा। इसके लिए निम्न Steps को perform करेंगे—

1. New Node बनायेंगे।
2. New Node के information field में Data value enter करेंगे।
3. New Node के linked field में NULL value insert करेंगे।
4. list का Pointer head इस नये node को Point करेगा।

यदि list Empty नहीं है तो नये node को list के सबसे आगे add कर दिया जायेगा। इसके लिए निम्न steps को follow करेंगे—

1. New node बनायेंगे।
2. New node के Information Part में Data Value enter करेंगे।
3. New node के Link Part में Head (list Pointer) जो कि list के First Item को point कर रहा है) का Address डालेंगे।
4. अब Head New Node को Point करेगा।



**Algorithm** (Inserting a Node at the Beginning) :

Insert - First [START, ITEM]

1. [Check for overflow?]

if PTR = NULL, then

Print overflow

Exit

else

PTR = (Node \*) malloc (Size of (node))

End if

2. Set PTR → INFO = Item

3. Set PTR → NEXT = START

4. Set START = PTR



**2. Insertion at the end of the list**—यदि किसी element को list के end में insert करना हो तो सबसे पहले list empty है या नहीं condition को test किया जाता है। यदि list empty है तो नया Node list का First node भी होगा और list का Last node भी होगा। अतः निम्न Steps को perform करेंगे—

1. New node बनायेंगे।
2. New node की Information field में Data value insert करेंगे।
3. New node के Link Part में NULL डालेंगे।
4. अब Head Pointer New node को point करेगा।

यदि list empty नहीं है तो list को traverse कर last element तक पहुँचेंगे और बाद में last में नये node को insert कर दिया जायेगा। इसके लिए निम्न Steps को follow करेंगे—

1. New Node बनायेंगे।
2. New Node की Information Part/field में Data value insert करेंगे।
3. New Node के link field में NULL डालेंगे।
4. अब Head Pointer के अलावा एक नये Pointer की आवश्यकता होगी जो list के last node तक जायेगा।
5. List के last node के link field में नये Node का Address डालेंगे।

**Algorithm** (Insert a node at the end) :

**Step 1 :** [Check for overflow]

if PTR = NULL, then

Print overflow

Exit.

else

PTR = (Node \*) malloc (size of (node))

end if.

**Step 2 :** Set PTR → info = Item

**Step 3 :** Set PTR → Next = NULL

**Step 4 :** if START = NULL and if then set START = P

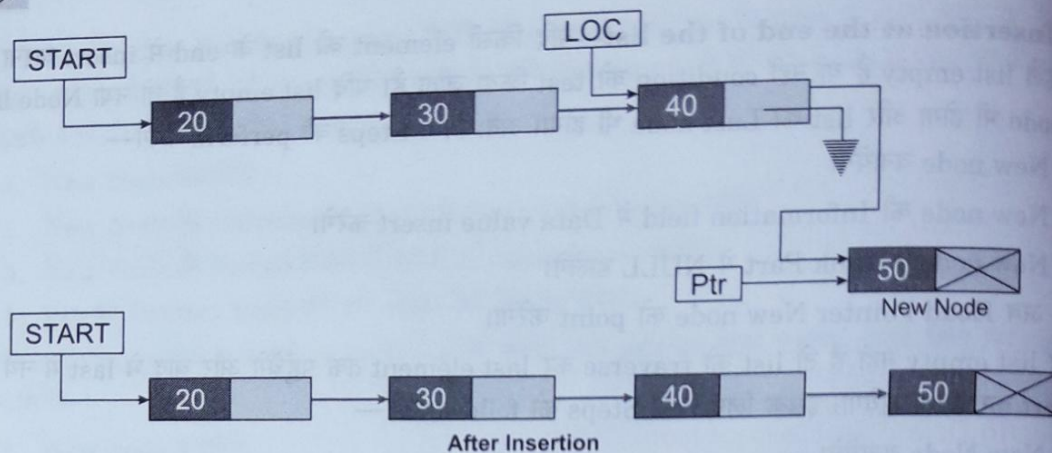
**Step 5 :** Set LOC = Start

**Step 6 :** Repeat Step 7 until LOC → next != NULL

**Step 7 :** Set LOC = LOC → next

**Step 8 :** Set LOC = next = P

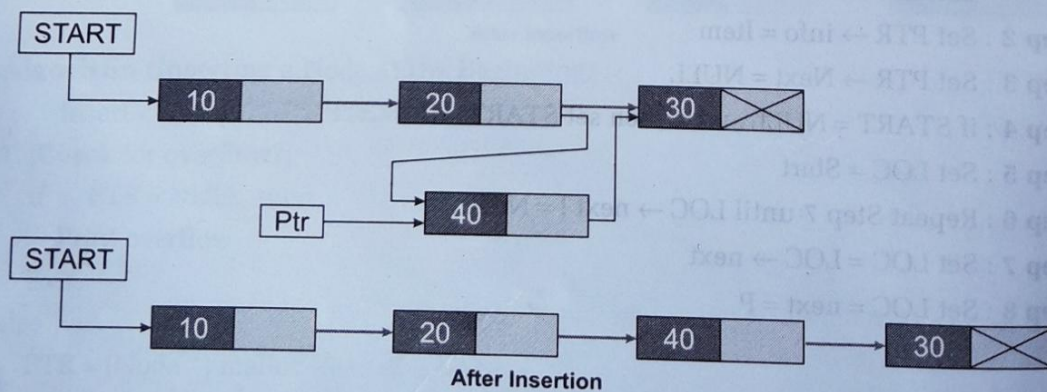




**3. Insertion at the Specified Position**—यदि नये node को किसी दिये गये element के बाद insert करना हो तो सबसे पहले उस location को Find out किया जायेगा जहाँ node को Add करना है और इसके बाद list में नये node को Add कर दिया जायेगा। इसके लिए निम्न Steps को perform करते हैं—

1. New node बनायेंगे।
2. New Node की information field/part में Data Value insert करेंगे।
3. एक Temporary Pointer लेंगे जो list के इस element को Point करेगा जिसके बाद नया node Add करना है।
4. Temporary Pointer NULL तक पहुँच जाता है अतः list में item नहीं है जिसके बाद नये Node को Add करना है।
5. यदि temporary Pointer इस item को ढूँढ़ लेता है तो निम्न Steps को follow करेंगे—
  - I. नये node के link field में temporary Pointer के link part को डाल देंगे।
  - II. Temporary Pointer के link field में New Node को डाल देंगे।

**Example :**



### Delete operation on a Singly linked list :

किसी भी element को list से Delete करने के लिए सबसे पहले Pointer को Properly set किया जाता है। बाद में इस Momory को Free कर दिया जाता है जिसे node द्वारा use में लिया जा रहा था। List को 3 जगहों से Delete किया जा सकता है—



1. List के Beginning से
2. List के End से
3. List के किसी दिये गये Location से।

### 1. Delete an Element from the beginning of list

जब list के first element को Delete करना हो तो इसके लिए निम्न Steps को perform किया जाता है—

- I. Head की Value को किसी temporary Variable में Assign करना।
- II. अब Head के next part को Head में Assign कर दिया जायेगा।
- III. जो Memory ptr द्वारा Point की जा रही है इसे Deallocate कर दिया जायेगा।

#### Algorithm to Delete an Element from Beginning :

Step 1. [Check for underflow]

if START = NULL, then

Print (linked list Empty)

Exit

end if

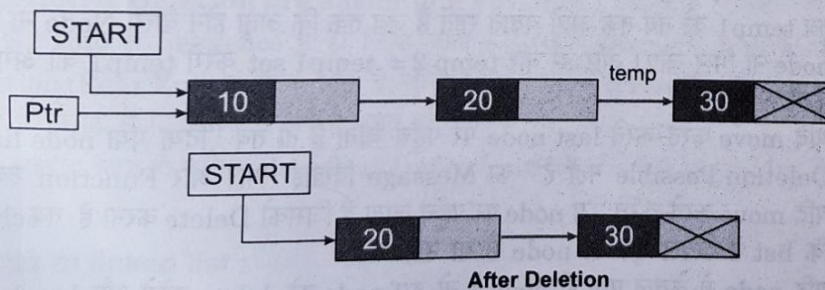
Step 2. Set PTR = START

Step 3. Set START = START → next

Step 4. Print Element Deleted is, Ptr → info

Step 5. free (ptr).

#### Example :



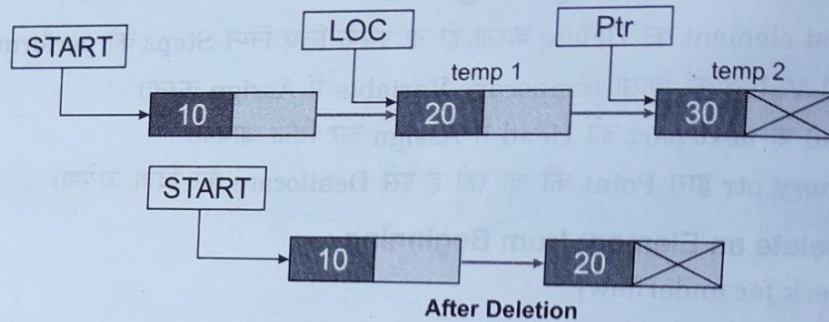
### Delete from the end of the list

किसी भी element को list के end से Delete करने के लिए सबसे पहले List को second last element तक traverse किया जायेगा। इसके बाद last element को Delete करने के लिए निम्न Steps को follow किया जायेगा—

1. यदि Head NULL है तो list में कोई Data item नहीं है अतः किसी Data item को Delete नहीं किया जा सकता।
2. एक temporary Pointer लेंगे जिसमें head की value को Assign करेंगे।
3. यदि Head का link Part NULL है तो list में एक item है जिसे Delete करना है।
4. इसे Delete करने के लिए head में NULL Assign कर देंगे और Pointer ptr को free कर देंगे।



5. यदि list में एक से ज्यादा items हैं अतः पहले NULL तक जायेंगे और एक Pointer last Node के पहले वाले Node को Point करेगा।
6. II<sup>nd</sup> last node के link Part में NULL Value assign कर देंगे।
7. वह Pointer जो last Node को Point कर रहा था उस free कर देंगे।

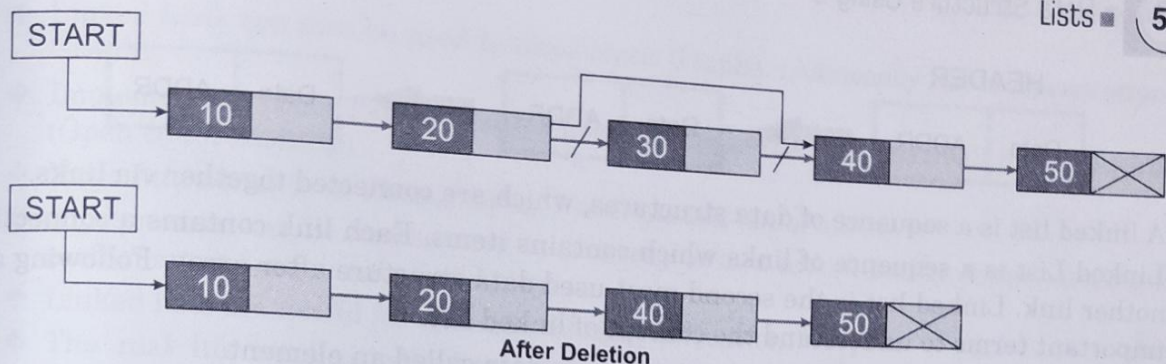


### Delete an Element from Specified Position

जब किसी दिये गये Element के बाद वाले element को Delete करना हो तो सबसे पहले location को find out किया जायेगा जिसके बाद वाले Element को Delete करना है इसके लिए निम्न steps को follow करते हैं—

- Step 1 :** सबसे पहले check करेंगे कि list empty है कि नहीं।
- Step 2 :** यदि यह (list) empty है तो यह message दिखाई देगा कि list खाली है और deletion नहीं हो सकता। और Function रुक जायेगा।
- Step 3 :** और यदि list खाली नहीं है तो temp1 और temp 2 दो Pointer node ले हैं और temp1 list के Head को assign कर देते हैं।
- Step 4 :** तब temp1 को तब तक आगे बढ़ाते रहते हैं जब तक कि अतू होने वाला Node ना मिल जाए या last node ना मिल जाए। और हर बार temp 2 = temp1 set करेंगे temp1 को अगले हद्द पर move करने से पहले।
- Step 5 :** यदि move करते-करते last node पर पहुँच जाता है तो तब “दिया गया node list में नहीं है। और Deletion Possible नहीं है” का Message दिखाई देगा। और Function रुक जायेगा।
- Step 6 :** यदि move करते-करते उस node पर पहुँच जाता है जिसको Delete करना है तब check किया जाता है कि list में केवल एक ही node है या नहीं।
- Step 7 :** यदि node में केवल एक ही node है तो उस node को delete करेंगे और head = NULL set कर देंगे और temp1 को Delete कर देंगे।
- Step 8 :** यदि list में कई सारे nodes हैं तो check करते हैं कि temp1 list का पहला node है तब temp1 = head set कर देंगे।
- Step 9 :** यदि temp1 पहला node है तब head को next node पर स्दन करेंगे (head = head → next) और Delete temp1.
- Step 10 :** यदि temp1 पहला node नहीं है तब कहीं ये last node तो नहीं है list का (temp1 → next == NULL)।
- Step 11 :** यदि temp1 last node है तब temp2 → next = NULL और Delete temp1.
- Step 12 :** यदि temp1, First node नहीं है और ना ही last node, तब temp2 → next = temp1 → next set करेंगे और temp1 को delete करेंगे।





**प्रश्न 3. Linked list के Advantages और Disadvantages को बताइये।**

**उत्तर—**Linked list के Advantages और Disadvantages निम्न प्रकार हैं—

### Advantages of linked list

Linked list के बहुत लाभ (advantages) हैं उनमें से कुछ निम्न हैं—

1. **Dynamic Data Structure :** Stack और Queue को declare करते समय उनकी size define करना पड़ता है जबकि linked list जरूरत पड़ने पर घटाया या बढ़ाया जा सकता है।
2. **Efficient Memory utilization :** जो चीजें limited होती हैं उनका utilization efficient ही होना चाहिए। ठीक इसी प्रकार computer में Memory limited होती है जिसका efficient use होना बहुत जरूरी है। Linked list में Declaration के समय जो size दिया जाता है इसे घटाया या बढ़ाया जा सकता है। जिस Data Element का प्रयोग नहीं किया जाना है इस element को linked list से remove कर दिया जाता है ताकि बचे हुए Memory को किसी दूसरे काम में लिया जा सके।
3. **Insertion & Deletion are easier and efficient :** Stack में Insertion और Deletion एक ओर से (Top) से Sequence में ही किया जाता है। जबकि Queue में Insertion (Front end) और Deletion (Rear End) से किया जाता है, बीच से नहीं। जबकि linked list में Insertion और Deletion किसी भी जगह से किया जा सकता है बहुत ही आसानी से।
4. इसके अलावा बहुत ही कठिन (Complex) Applications को आसानी से linked list के द्वारा use किया जाता है।

### Disadvantages of linked list :

1. **More Memory :** अगर linked list में number of nodes बहुत ज्यादा हैं तो इस Case में linked list को store होने के लिए बहुत ज्यादा Memory की जरूरत पड़ेगी।
2. अगर linked list में node ज्यादा हैं तो इनमें से किसी node को search करने में समय (time) ज्यादा लगेगा।

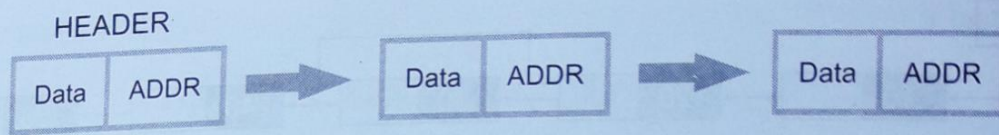
### Introduction to Linked Lists

Linked List is a very commonly used linear data structure which consists of group of nodes in a sequence.

Each node holds its own **data** and **address of the next node** hence forming a chain like structure.

Linked Lists are used to create trees and graphs.





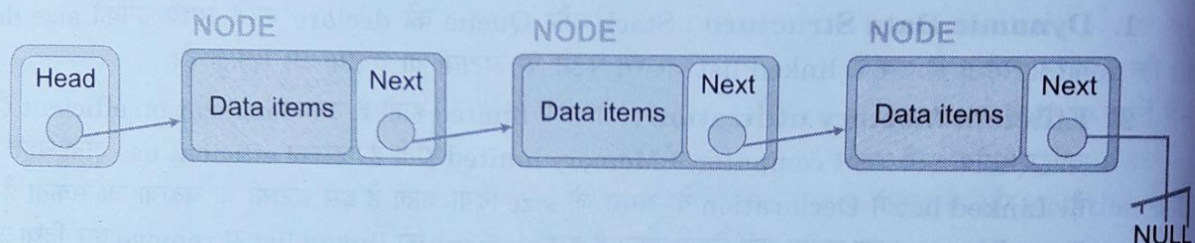
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of linked list.

- ❖ **Link** - Each link of a linked list can store a data called an element.
- ❖ **Next** - Each link of a linked list contains a link to the next link called Next.
- ❖ **Linked List** - A Linked List contains the connection link to the first link called First.

### Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered :

- ❖ Linked List contains a link element called first.
- ❖ Each link carries a data field(s) and a link field called next.
- ❖ Each link is linked with its next link using its next link.
- ❖ Last link carries a link as null to mark the end of the list.

### Advantages of Linked Lists

- ❖ They are dynamic in nature which allocates the memory when required.
- ❖ Insertion and deletion operations can be easily implemented.
- ❖ Stacks and queues can be easily executed.
- ❖ Linked List reduces the access time.

### Disadvantages of Linked Lists

- ❖ The memory is wasted as pointers require extra memory for storage.
- ❖ No element can be accessed randomly; it has to access each node sequentially.
- ❖ Reverse Traversing is difficult in linked list.

### Applications of Linked Lists

#### Applications of Linked List data structure

There are various applications of doubly linked list in the real world. Some of them can be listed as :

- ❖ Linked Lists can be used to implement Stacks, Queues.



- ❖ Linked Lists can also be used to implement Graphs. (Adjacency list representation of Graph).
- ❖ Implementing Hash Tables : Each Bucket of the hash table can itself be a linked list. (Open chain hashing).
- ❖ Undo functionality in Photoshop or Word. Linked list of states.
- ❖ A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term.
- ❖ Linked lists are useful for dynamic memory allocation.
- ❖ The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running.
- ❖ All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The operating System keeps on iterating over the linked list until all the applications are completed.
- ❖ Linked lists are used when the quantity of data is not known prior to execution.

## Types of Linked Lists

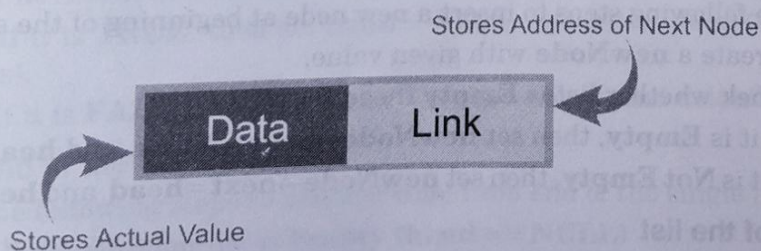
1. Singly linked list
2. Doubly linked list
3. Circular linked list
4. Circular doubly linked list

Simply a list is a sequence of data, and linked list is a sequence of data linked with each other. the formal definition of a single linked list is as follows :

**Single linked list is a sequence of elements in which every element has link to its next element in the sequence.**

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data and next. The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

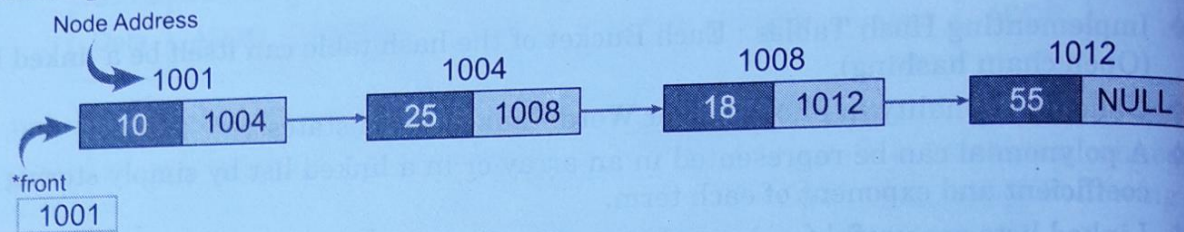
The graphical representation of a node in a single linked list is as follows :



### Note :

- ❖ In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").
- ❖ Always next part (reference part) of the last node must be NULL.



**Example :****Operations**

In a single linked list we perform the following operations :

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to set up empty list. First perform the following steps before implementing actual operations :

- ❖ **Step 1** : Include all the **header files** which are used in the program.
- ❖ **Step 2** : Declare all the **user defined** functions.
- ❖ **Step 3** : Define a **Node** structure with two members **data** and **next**.
- ❖ **Step 4** : Define a Node Pointer '**head**' and set it to **NULL**.
- ❖ **Step 5** : Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

**Insertion**

In a single linked list, the insertion operation can be performed in three ways. They are as follows :

1. Inserting at Beginning of the list
2. Inserting at End of the list
3. Inserting at Specific location in the list

**Inserting at Beginning of the list**

We can use the following steps to insert a new node at beginning of the single linked list

- ❖ **Step 1** : Create a **newNode** with given value.
- ❖ **Step 2** : Check whether list is **Empty** (**head == NULL**)
- ❖ **Step 3** : If it is **Empty**, then set **newNode→next = NULL** and **head = newNode**.
- ❖ **Step 4** : If it is **Not Empty**, then set **newNode→next = head** and **head = newNode**.

**Inserting at End of the list**

We can use the following steps to insert a new node at end of the single linked list :

- ❖ **Step 1** : Create a **newNode** with given value and **newNode → next** as **NULL**.
- ❖ **Step 2** : Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 3** : If it is **Empty**, then set **head = newNode**.
- ❖ **Step 4** : If it is **Not Empty**, then define a node pointer **temp** and initialize with **head**.
- ❖ **Step 5** : Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).



- ❖ **Step 6 :** Set **temp** → **next** = **newNode**.

### Inserting at Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list :

- ❖ **Step 1 :** Create a **newNode** with given value.
- ❖ **Step 2 :** Check whether list is **Empty** (**head** == **NULL**)
- ❖ **Step 3 :** If it is **Empty** then, set **newNode** → **next** = **NULL** and **head** = **newNode**.
- ❖ **Step 4 :** If it is **Not Empty**, then define a node pointer **temp** and initialize with **head**.
- ❖ **Step 5 :** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp** → **data** is equal to **location**, here location is the node value after which we want to insert the **newNode**).
- ❖ **Step 6 :** Every time check whether **temp** is reached to last node or not. If it is reached to last node, then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.
- ❖ **Step 7 :** Finally, set '**newNode** → **next** = **temp** → **next**' and '**temp** → **next** = **newNode**'.

### Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows :

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

#### Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list :

- ❖ **Step 1 :** Check whether list is **Empty** (**head** == **NULL**).
- ❖ **Step 2 :** If it is **Empty**, then display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- ❖ **Step 3 :** If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **head**.
- ❖ **Step 4 :** Check whether list is having only one node (**temp** → **next** == **NULL**)
- ❖ **Step 5 :** If it is **TRUE**, then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions).
- ❖ **Step 6 :** If it is **FALSE**, then set **head** = **temp** → **next**, and delete **temp**.

#### Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list :

- ❖ **Step 1 :** Check whether list is **Empty** (**head** == **NULL**)
- ❖ **Step 2 :** If it is **Empty**, then display List is '**Empty!!! Deletion is not possible**' and terminate the function.
- ❖ **Step 3 :** If it is **Not Empty**, then define two node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- ❖ **Step 4 :** Check whether list has only one Node (**temp1** → **next** == **NULL**)
- ❖ **Step 5 :** If it is **TRUE**, then set **head** = **NULL** and delete **temp1**. And terminate the function (Setting **Empty** list condition).



- ❖ **Step 6 :** If it is FALSE, then set **'temp2 = temp1'** and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next == NULL**).
- ❖ **Step 7 :** Finally, set **temp2 → next = NULL** and delete **temp1**.

### Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list :

- ❖ **Step 1 :** Check whether list is **Empty (head == NULL)**.
- ❖ **Step 2 :** If it is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.
- ❖ **Step 3 :** If it is **Not Empty**, then define two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**.
- ❖ **Step 4 :** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set **'temp2 = temp1'** before moving the **'temp1'** to its next node.
- ❖ **Step 5 :** If it is reached to the last node, then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.
- ❖ **Step 6 :** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not.
- ❖ **Step 7 :** If list has only one node and that is the node to be deleted, then set **head = NULL** and delete **temp1 (free(temp1))**.
- ❖ **Step 8 :** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- ❖ **Step 9 :** If **temp1** is the first node, then move the **head** to the next node (**head = head → next**) and delete **temp1**.
- ❖ **Step 10 :** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
- ❖ **Step 11 :** If **temp1** is last node then set **temp2 → next = NULL** and delete **temp1 (free(temp1))**.
- ❖ **Step 12 :** If **temp1** is not first node and not last node, then set **temp2 → next = temp1 → next** and delete **temp 1 (free(temp 1))**.

### Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list :

- ❖ **Step 1 :** Check whether list is **Empty (head == NULL)**.
- ❖ **Step 2 :** If it is **Empty**, then display '**List is Empty!!!**' and terminate the function.
- ❖ **Step 3 :** If it is **Not Empty**, then define a Node Pointer **'temp'** and initialize with **head**.
- ❖ **Step 4 :** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node.
- ❖ **Step 5 :** Finally display **temp → data** with arrow pointing to **NULL (temp → data ---> NULL)**.

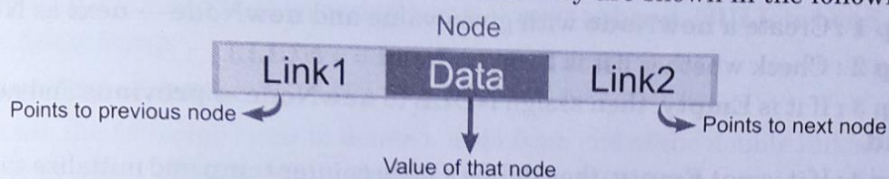


### Doubly linked list :

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows :

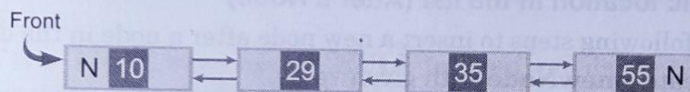
**Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.**

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure :



Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

#### Example :



#### NOTE :

- ❖ In double linked list, the first node must be always pointed by **head**.
- ❖ Always the previous field of the first node must be **NULL**.

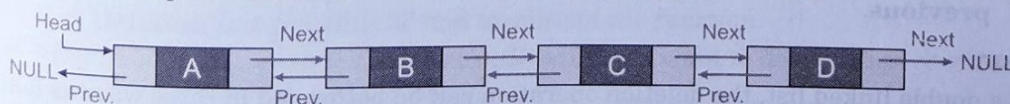
#### Operations :

In a double linked list, we perform the following operations :

1. Insertion
2. Deletion
3. Display

#### Inserting a node in singly Linked List

A **Doubly Linked List (DLL)** contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



1. Inserting at beginning of the list
2. Inserting at End of the list
3. Inserting at Specific location in the list



### Inserting at Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list :

- ❖ **Step 1** : Create a **newNode** with given value and **newNode** → **previous** as **NULL**.
- ❖ **Step 2** : Check whether list is **Empty** (**head** == **NULL**).
- ❖ **Step 3** : If it is **Empty**, then assign **NULL** to **newNode** → **next** and **newNode** to **head**.
- ❖ **Step 4** : If it is **not Empty**, then assign **head** to **newNode** → **next** and **newNode** to **head**.

### Inserting at End of the list

We can use the following steps to insert a new node at end of the double linked list :

- ❖ **Step 1** : Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- ❖ **Step 2** : Check whether list is **Empty** (**head** == **NULL**).
- ❖ **Step 3** : If it is **Empty**, then assign **NULL** to **newNode** → **previous** and **newNode** to **head**.
- ❖ **Step 4** : If it is **not Empty**, then define a node pointer **temp** and initialize with **head**.
- ❖ **Step 5** : Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- ❖ **Step 6** : Assign **newNode** to **temp** → **next** and **temp** to **newNode** → **previous**.

### Inserting at Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list :

- ❖ **Step 1** : Create a **newNode** with given value.
- ❖ **Step 2** : Check whether list is **Empty** (**head** == **NULL**).
- ❖ **Step 3** : If it is **Empty**, then assign **NULL** to **newNode** → **previous** & **newNode** → **next** and **newNode** to **head**.
- ❖ **Step 4** : If it is **not Empty**, then define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- ❖ **Step 5** : Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the **newNode**).
- ❖ **Step 6** : Every time check whether **temp1** is reached to the last node. If it is reached to the last node, then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp1** to next node.
- ❖ **Step 7** : Assign **temp1** → **next** to **temp2**, **newNode** to **temp1** → **next**, **temp1** to **newNode** → **previous**, **temp2** to **newNode** → **next** and **newNode** to **temp2** → **previous**.

### Deletion

In a double linked list, the deletion operation can be performed in three ways as follows :

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node



### Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list :

- ❖ **Step 1 :** Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 2 :** If it is **Empty**, then display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- ❖ **Step 3 :** If it is not Empty, then, define a Node pointer '**temp**' and initialize with **head**.
- ❖ **Step 4 :** Check whether list is having only one node (**temp → previous** is equal to **temp → next**).
- ❖ **Step 5 :** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions).
- ❖ **Step 6 :** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

### Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list :

- ❖ **Step 1 :** Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 2 :** If it is **Empty**, then display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- ❖ **Step 3 :** If it is not Empty, then define a Node pointer '**temp**' and initialize with **head**.
- ❖ **Step 4 :** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**).
- ❖ **Step 5 :** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition).
- ❖ **Step 6 :** If it is **FALSE**, then keep moving **temp** unit it reaches to the last node in the list. (until **temp → next** is equal to **NULL**).
- ❖ **Step 7 :** Assign **NULL** to **temp → previous → next** and delete **temp**.

### Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list :

- ❖ **Step 1 :** Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 2 :** If it is **Empty**, then display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- ❖ **Step 3 :** If it is not Empty, then define a Node pointer '**temp**' and initialize with **head**.
- ❖ **Step 4 :** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- ❖ **Step 5 :** If it is reached to the last node, then display '**Given node not found in the list! Deletion not possible!!!**' and terminate the function.
- ❖ **Step 6 :** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not.
- ❖ **Step 7 :** If list has only one node and that is the node which is to be deleted, then set **head** to **NULL** and delete **temp** (**free(temp)**).
- ❖ **Step 8 :** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).



- ❖ **Step 9 :** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- ❖ **Step 10 :** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- ❖ **Step 11 :** If **temp** is the last node, then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- ❖ **Step 12 :** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

### Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list :

- ❖ **Step 1 :** Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 2 :** If it is **Empty**, then display '**List is Empty!!!**' and terminate the function.
- ❖ **Step 3 :** If it is not **Empty**, then define a Node pointer '**temp**' and initialize with **head**.
- ❖ **Step 4 :** Display '**NULL**'.
- ❖ **Step 5 :** Keep displaying **temp → data** with an arrow (**<== ==>**) until **temp** reaches to the last node.
- ❖ **Step 6 :** Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data → NULL**).

### Applications/Uses of Doubly Linked List in Real Life

There are various applications of doubly linked list in the real world. Some of them can be listed as :

- ❖ Doubly linked list can be used in navigation systems where both front and back navigation is required.
- ❖ It is used by browsers to implement backward and forward navigation of visited web pages i.e. **back** and **forward** button.
- ❖ It is also used by various applications to implement **Undo** and **Redo** functionality.

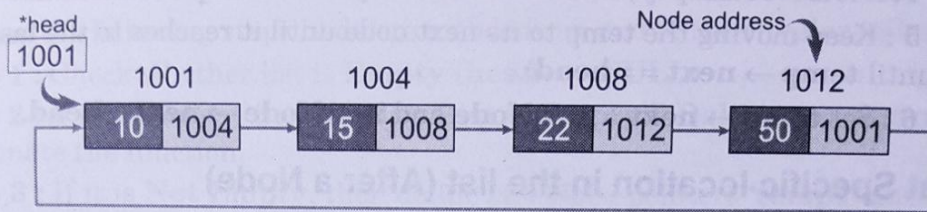
### Circular link list :

In single linked list, every node points to its next node in the sequence and the last node points **NULL**. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

**Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.**

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.



**Example :**

In a circular linked list, we perform the following operations :

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to set up empty list. First perform the following steps before implementing actual operations.

- ❖ **Step 1 :** Include all the **header files** which are used in the program.
- ❖ **Step 2 :** Declare all the **user defined** functions.
- ❖ **Step 3 :** Define a **Node** structure with two members **data** and **next**.
- ❖ **Step 4 :** Define a Node pointer '**head**' and set it to **NULL**.
- ❖ **Step 5 :** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

**Insertion**

In a circular linked list, the insertion operation can be performed in three ways. They are as follows :

1. Inserting at Beginning of the list
2. Inserting at End of the list
3. Inserting at Specific location in the list

**Inserting at beginning of the list**

We can use the following steps to insert a new node at beginning of the circular linked list :

- ❖ **Step 1 :** Create a **newNode** with given value.
- ❖ **Step 2 :** Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 3 :** If it is **Empty**, then set **head = newNode** and **newNode → next = head**.
- ❖ **Step 4 :** If it is **Not Empty**, then define a Node Pointer '**temp**' and initialize with '**head**'.
- ❖ **Step 5 :** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next == head**').
- ❖ **Step 6 :** Set '**newNode → next = head**', '**head = newNode**' and '**temp → next = head**'.

**Inserting at End of the list**

We can use the following steps to insert a new node at end of the circular linked List :

- ❖ **Step 1 :** Create a **newNode** with given value.
- ❖ **Step 2 :** Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 3 :** If it is **Empty**, then set **head = newNode** and **newNode → next = head**.



- ❖ **Step 4 :** If it is **Not Empty**, then define a node pointer **temp** and initialize with **head**.
- ❖ **Step 5 :** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next == head**).
- ❖ **Step 6 :** Set **temp → next = newNode** and **newNode → next = head**.

### Inserting at Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list.

- ❖ **Step 1 :** Create a **newNode** with given value.
- ❖ **Step 2 :** Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 3 :** If it is **Empty**, then set **head = newNode** and **newNode → next = head**.
- ❖ **Step 4 :** If it is **Not Empty**, then define a node pointer **temp** and initialize with **head**.
- ❖ **Step 5 :** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp1 → data** is equal to **location**, here **location** is the node value after which we want to insert the **newNode**).
- ❖ **Step 6 :** Every time check whether **temp** is reached to the last node or not. If it is reached to last node, then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.
- ❖ **Step 7 :** If **temp** is reached to the exact node after which we want to insert the **newNode** then check whether it is last node (**temp → next == head**).
- ❖ **Step 8 :** If **temp** is last node then set **temp → next = newNode** and **newNode → next = head**.
- ❖ **Step 9 :** If **temp** is not last node, then set **newNode → next = temp → next** and **temp → next = newNode**.

### Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows :

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

#### Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list.

- ❖ **Step 1 :** Check whether list is **Empty** (**head == NULL**).
- ❖ **Step 2 :** If it is **Empty**, then display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- ❖ **Step 3 :** If it is **Not Empty**, then define two Node pointers '**temp1**' and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- ❖ **Step 4 :** Check whether list is having only one node (**temp1 → next == head**).
- ❖ **Step 5 :** If it is **TRUE**, then set **head = NULL** and delete **temp1** (Setting **Empty** list conditions).
- ❖ **Step 6 :** If it is **FALSE**, move the **temp1** until it reaches to the last node. (until **temp1 → next == head**).
- ❖ **Step 7 :** Then set **head = temp2 → next**, **temp1 → next = head** and delete **temp2**.



### Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list :

- ❖ **Step 1** : Check whether list is **Empty** ( $\text{head} == \text{NULL}$ ).
- ❖ **Step 2** : If it is **Empty**, then display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- ❖ **Step 3** : If it is **Not Empty**, then define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with head.
- ❖ **Step 4** : Check whether list has only one Node ( $\text{temp1} \rightarrow \text{next} == \text{head}$ ).
- ❖ **Step 5** : If it is **TRUE**, then set  $\text{head} = \text{Null}$  and delete **temp1**. And terminate from the function. (Setting **Empty** list condition).
- ❖ **Step 6** : If it is **FALSE**, then set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list (until  $\text{temp1} \rightarrow \text{next} = \text{head}$ ).
- ❖ **Step 7** : Set  $\text{temp2} \rightarrow \text{next} = \text{head}$  and delete **temp1**.

### Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list :

- ❖ **Step 1** : Check whether list is **Empty** ( $\text{head} == \text{NULL}$ ).
- ❖ **Step 2** : If it is **Empty**, then display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- ❖ **Step 3** : If it is **Not empty**, then define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with head.
- ❖ **Step 4** : Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- ❖ **Step 5** : If it is reached to the last node, then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.
- ❖ **Step 6** : If it is reached to the exact node which we want to delete, then check whether list is having only one node ( $\text{temp1} \rightarrow \text{next} == \text{head}$ ).
- ❖ **Step 7** : If list has only one node and that is the node to be deleted, then set  $\text{head} = \text{NULL}$  and delete **temp1** ( $\text{free}(\text{temp1})$ ).
- ❖ **Step 8** : If list contains multiple nodes, then check whether **temp1** is the first node in the list ( $\text{temp1} == \text{head}$ ).
- ❖ **Step 9** : If **temp1** is the first node, then set  $\text{temp2} = \text{head}$  and keep moving **temp2** to its next node until **temp2** reaches to the last node. The set  $\text{head} = \text{head} \rightarrow \text{next}$ ,  $\text{temp2} \rightarrow \text{next} = \text{head}$  and delete **temp1**.
- ❖ **Step 10** : If **temp1** is not first node, then check whether it is last node in the list ( $\text{temp1} \rightarrow \text{next} == \text{head}$ ).
- ❖ **Step 11** : If **temp1** is last node, then set  $\text{temp2} \rightarrow \text{next} = \text{head}$  and delete **temp1** ( $\text{free}(\text{temp1})$ ).
- ❖ **Step 12** : If **temp1** is not first node and not last node then set  $\text{temp2} \rightarrow \text{next} = \text{temp1} \rightarrow \text{next}$  and delete **temp1** ( $\text{free}(\text{temp1})$ ).



### Displaying a Circular Linked List

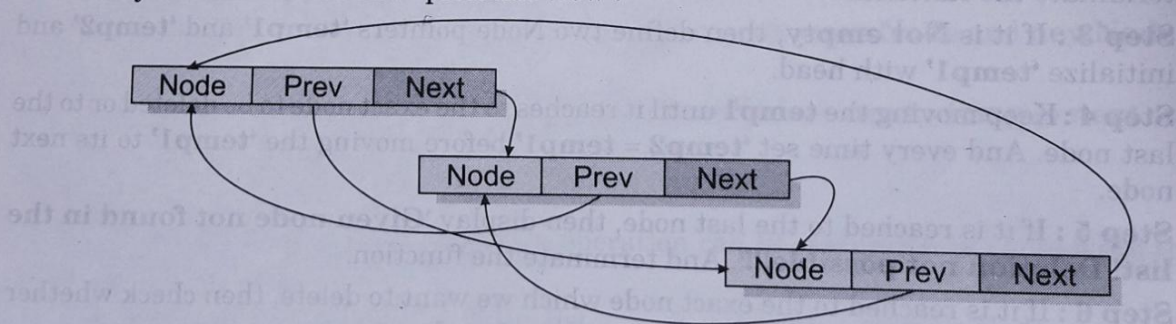
We can use the following steps to display the elements of a circular linked list :

- ❖ **Step 1 :** Check whether list is **Empty** ( $\text{head} == \text{NULL}$ ).
- ❖ **Step 2 :** If it is **Empty**, then display '**List is Empty!!!**' and terminate the function.
- ❖ **Step 3 :** If it is **Not Empty**, then define a Node Pointer '**temp**' and initialize with head.
- ❖ **Step 4 :** Keep displaying **temp**  $\rightarrow$  **data** with an arrow ( $\rightarrow$ ) until **temp** reaches to the last node.
- ❖ **Step 5 :** Finally display **temp**  $\rightarrow$  **data** with arrow pointing to **head**  $\rightarrow$  **data**.

### Doubly Circular linked list

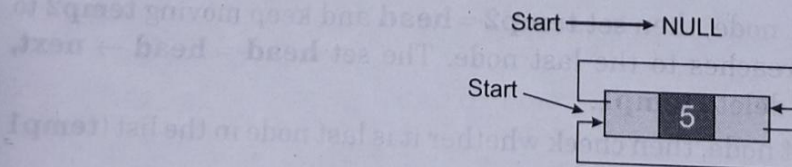
Doubly Circular linked list has both the properties of doubly linked list and circular linked list. Two consecutive elements are linked by previous and next pointer and the last node points to first node by next pointer and also the previous pointer of the head node points to the tail node. This two way pointer linking has eliminated all the shortcomings of all previous linked lists which have been discussed in the previous sections. Node traversal from any direction is possible and also jumping from head to tail or from tail to head is only one operation: head pointer previous is tail and also tail pointer next is head. Find the visual representation of the doubly circular linked list in the below figure.

Doubly circular linked list - pictorial view :



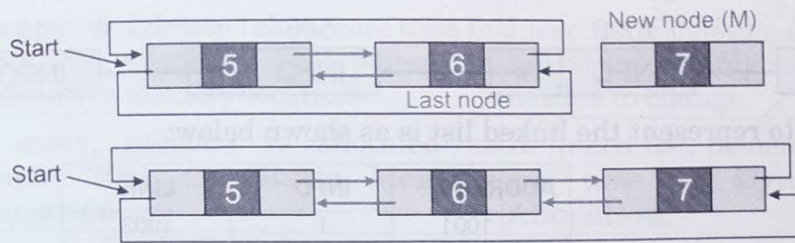
#### 1. Insertion at the end of list or in an empty list

- ❖ **Empty List ( $\text{start} = \text{NULL}$ ) :** A node (Say N) is inserted with data = 5, so previous pointer of N points to N and next pointer of N also points to N. But now start pointer points to the first node of the list.



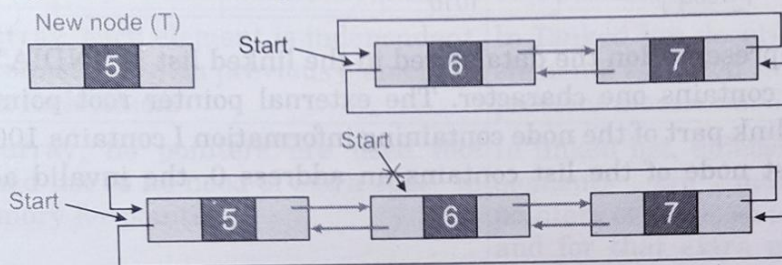
- ❖ **List initially contain some nodes, start points to first node of the List :** A node (Say M) is inserted with data = 7, so previous pointer of M points to last node, next pointer of M points to first node and last node's next pointer points to this M node and first node's previous pointer points to this M node.





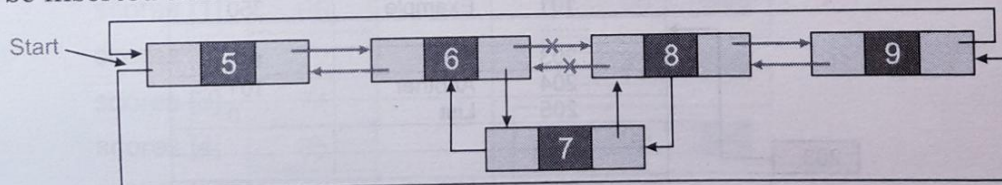
## 2. Insertion at the beginning of the list

To insert a node at the beginning of the list, create a node (Say T) with data = 5, T next pointer points to first node of the list, T previous pointer points to last node of the list, last node's next pointer points to this T node, first node's previous pointer also points this T node and at last don't forget to shift 'Start' pointer to this T node.



## 3. Insertion in between the nodes of the list

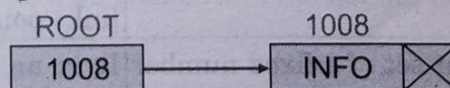
To insert a node in between the list, two data values are required one after which new node will be inserted and another is the data of the new node.



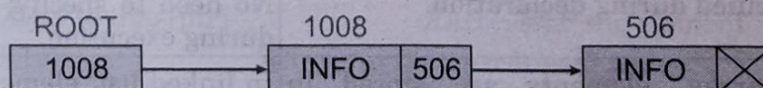
## Memory Representation of Linked List

In memory, the linked list is stored in scattered cells (locations). The memory for each node is allocated dynamically means as and when required. So the Linked List can increase as per the user wish and the size is not fixed, it can vary.

Suppose first node of linked list is allocated with an address 1008. Its graphical representation looks like the figure shown below :

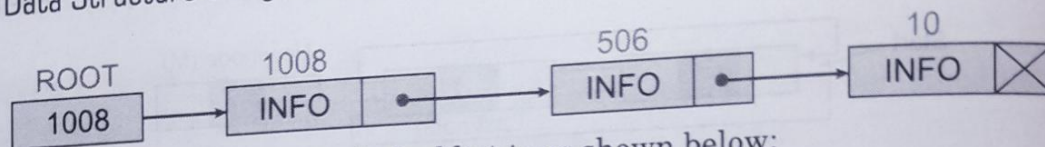


Suppose next node is allocated at an address 506, so the list becomes,



Suppose next node is allocated with an address with an address 10, the list becomes.





The other way to represent the linked list is as shown below:

| ADDRESS | INFO | LINK |
|---------|------|------|
| 1001    | I    | 1003 |
| 1002    |      |      |
| 1003    | A    | 0    |
| 1004    |      |      |
| 1005    | I    | 1007 |
| 1006    |      |      |
| 1007    | N    | 1009 |
| 1008    |      |      |
| 1009    | D    | 1001 |
| 1010    |      |      |

Diagram showing a pointer box labeled '1005' pointing to the row with address 1005 in the table.

In the above representation the data stored in the linked list is "INDIA", the information part of each node contains one character. The external pointer root points to first node's address 1005. The link part of the node containing information I contains 1007, the address of next node. The last node of the list contains an address 0, the invalid address or NULL address.

#### Example :

| ADDRESS | INFO    | LINK |
|---------|---------|------|
| .....   |         |      |
| .....   |         |      |
| 100     | Is      | 204  |
| 101     | Example | 350  |
| .....   |         |      |
| 203     | This    | 100  |
| 204     | Another | 101  |
| 205     | List    | 0    |
| .....   |         |      |
| 306     | Linked  | 205  |
| 307     |         |      |
| .....   |         |      |
| 350     | Of      | 306  |
| .....   |         |      |

Diagram showing a pointer box labeled '203' pointing to the row with address 203 in the table.

#### Difference between Array and Linked List :

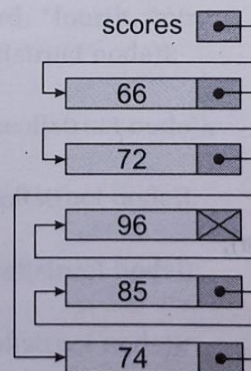
|                         | Array                                                                     | Linked List                                                                                                  |
|-------------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <b>Define</b>           | Array is a collection of elements having same data type with common name. | Linked list is an ordered collection of elements which are connected by links/pointers.                      |
| <b>Basic</b>            | It is a consistent set of a fixed number of data items.                   | It is an ordered set consisting of a variable number of data items.                                          |
| <b>Size</b>             | Specified during declaration.                                             | No need to specify; grow and shrink during execution.                                                        |
| <b>Memory Structure</b> | In array, elements are stored in consecutive manner in memory.            | In linked list, elements can be stored at any available place as address of node is stored in previous node. |



|                                 |                                                                                                       |                                                                                                                                                        |
|---------------------------------|-------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Insertion &amp; Deletion</b> | Insertion & deletion takes more time in array as elements are stored in consecutive memory locations. | Insertion & deletion are fast & easy in linked list as only value of pointer is needed to change.                                                      |
| <b>Memory Allocation</b>        | In array, memory is allocated at compile time <i>i.e.</i> <b>Static Memory Allocation.</b>            | In linked list, memory is allocated at run time <i>i.e.</i> <b>Dynamic Memory Allocation.</b>                                                          |
| Accessing the element           | Direct or randomly accessed, <i>i.e.</i> , Specify the array index or subscript.                      | Sequentially accessed, <i>i.e.</i> , Traverse starting from the first node in the list by the pointer.                                                 |
| Searching                       | Binary search and linear search                                                                       | linear search                                                                                                                                          |
| <b>Types</b>                    | Array can be <b>single dimensional</b> , two dimensional or <b>multidimensional</b> .                 | Linked list can be <b>singly</b> , <b>doubly</b> or <b>circular</b> linked list.                                                                       |
| <b>Dependency</b>               | In array, each element is independent, no connection with previous element or with its location.      | In Linked list, location or address of elements is stored in the link part of previous element/node.                                                   |
| <b>Extra Space</b>              | In array, no pointers are used like linked list so no need of extra space in memory for pointer.      | In linked list, adjacency between the elements are maintained using pointers or links, so pointers are used and for that extra memory space is needed. |

|            |        |
|------------|--------|
|            | scores |
| scores [1] | 66     |
| scores [2] | 72     |
| scores [3] | 74     |
| scores [4] | 85     |
| scores [5] | 96     |

Array representation



Linked list representation

## Comparison Chart

### Traversing in linked list :

It refers to an operation in which all elements of the list are accessed only once. Algorithm for traversing in Linked list is as follow :

1. Set ptr = Start
2. Repeat the steps 3 and 4 while ptr != Null
3. Apply PROCESS to info [ptr] //Accessing the element
4. Set ptr = link [ptr] //Going to the next linked node
5. Exit



```

#include<stdio.h>
#include<stdlib.h>
void main()
{
    struct node
    {
        int data;
        struct node *link;
    };
    struct node *first, *second, *third, *fourth, *ptr;
    first = (struct node*) malloc (sizeof (struct node));
    first->data = 10;
    second = (struct node*) malloc (sizeof (struct node));
    second->data = 30;
    third = (struct node*) malloc (sizeof (struct node));
    third->data = 40;
    fourth = (struct node*) malloc (sizeof (struct node));
    fourth->data = 60;
    ptr = (struct node*) malloc (sizeof (struct node));
    first->link = second;
    second->link = third;
    third->link = fourth;
    fourth->link = NULL;
    ptr = first;
    while (ptr != NULL)
    {
        printf("%d\n", ptr->data);
        ptr = ptr->link;
    }
}

```

## Searching

Consider that we are having a number of elements in a linked list and we want to confirm that whether it contains the desired element or not. For this we need to do the searching operation. There are two ways of doing so :

- # When the elements in list are unsorted
- # When the elements in list are sorted

### List is unsorted :

When the list is unsorted we need to access each element to find our data item. The advantage of using this method is that there is no need to sort the numbers. And the major disadvantage is that the number of comparisons in this method is pretty more hence its time consuming.



1. Set ptr = Start
2. Repeat the step 3 while ptr != Null
3. If item == info [ptr]                   //Comparing data item with the elements of list  
then set loc = ptr and print "search is successful" and exit else  
set ptr = link [ptr]                   //Going to the next linked node
4. Set loc = null and print "search is unsuccessful"
5. Exit.

**Example :**

//WAP to demonstrate searching,

```
#include<stdio.h>
#include<stdlib.h>
void main ( )
{
    struct node
    {
        int data;
        struct node *link;
    };
    int item, x;
    struct node *first, *second, *third, *fourth, *ptr;
    first=(struct node*)malloc(sizeof(struct node));
    first->data = 1;
    second=(struct node*)malloc(sizeof(struct node));
    second->data = 16;
    third=(struct node*)malloc (sizeof(struct node));
    third->data = 6;
    fourth=(strut node*)malloc (sizeof(struct node));
    fourth->data = 10;
    ptr = (struct node*)malloc (sizeof(struct node));
    first->link = second;
    second -> link = third;
    third -> link = fourth;
    fourth -> link = NULL;
    ptr = first;
    printf("Items in linked list are:\n");
    while(ptr != NULL)
    {
        printf("%d\n", ptr-> data);
        ptr = ptr -> link;
    }
    printf("Enter item to be searched:");
    scanf ("%d", &item);
    ptr = first;
```



```

while (ptr != NULL)
{
    If (item == ptr->data)
    {
        x = 1;
        printf("Search is successful\n");
        break;
    }
    else
        ptr = ptr->link;
}
If (x != 1)
{
    Printf("Search is unsuccessful\n");
}
}

```

#### List is sorted (descending order) :

When the list is sorted there is no need to access **each** element to find our data item. Suppose that our list is sorted in descending order, then we will start comparing from the first node and go on comparing till the data item is smaller than the info of that node. The advantage of this method is that the total number of comparisons is decreased hence the time taken to search is also decreased. The major disadvantage of using this method is that it is hard to maintain a sorted list *i.e.* whenever some insertion or deletion operation is to be performed then the list should remain sorted. Algorithm for this is :

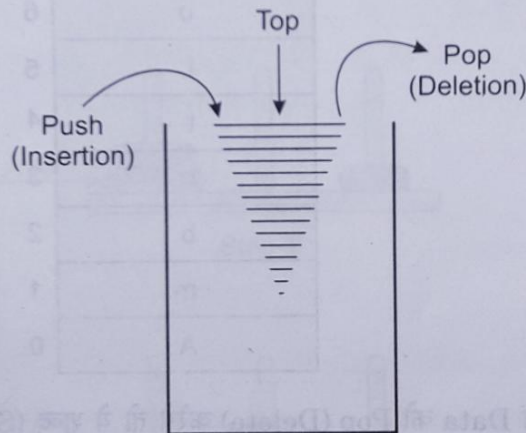
1. Set ptr = Start
2. Repeat the step 3 while ptr != Null
3. if item == info [ptr]      //Comparing data item with the elements of list  
     then set loc = ptr and print "search is successful" and exit  
     else if (item < info[ptr])  
         set ptr = link[ptr]      //Going to the next linked node  
     else if (item > info[ptr])  
         Set loc=Null and print "search is unsuccessful" and exit
4. Set loc=null and print "search is unsuccessful"
5. Exit.



# Stacks and Queues

**प्रश्न 1. Stack क्या होता है? इसके Applications का वर्णन करें।**

**उत्तर—**Stack एक Non-primitive linear data structure होता है जिसमें किसी नये Data item को जोड़ना या पहले से उपस्थित Data item को Delete करने का काम Stack के एक End (छोर) से किया जाता है जैसे Stack का Top कहते हैं। इसी Top से सभी Data items का Insertion और Deletion किया जाता है। Stack में जो Data item last में Add होता है वह Deletion के समय सबसे पहले Delete होता है। इसी वजह से इसे Last-In-First-Out (LIFO) प्रकार का List भी कहा जाता है।



**Example 1 :** Stack का एक Common Example है Stack of Plates in a marriage party or coin stacker. जहाँ पर fresh प्लेट Top पर रखा जाता है और top से ही Plates remove किया जाता है।

**Example 2 :** यदि हम Biscuits या poppins खाते हैं पर दोनों के एक छोर (End) को Open करके एक-एक करके Biscuits और Poppins को निकालते हैं। यहाँ पर भी Stack की Property को use किया जाता है।

## Basic Features of Stack

1. Stack एक ही तरह के Data types का sequential collection होता है।
2. Stack एक last-in-first-out प्रकार का Data Structure है।

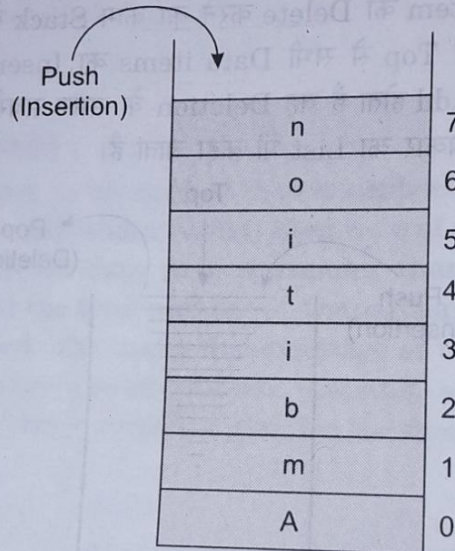


3. Stack में Data item को insert करने के लिए Push ( ) function का प्रयोग (use) किया जाता है जबकि Delete करने के लिए Pop ( ) function का। दोनों insertion और Deletion करने के लिए केवल एक छोटा (End) का प्रयोग किया जाता है जिसको Top of the Stack कहते हैं।
4. यदि किसी Stack की size पूरी तरह Full है तो Data item को Add (Insertion) करते समय overflow State बन जाता है जबकि एक ऐसा State भी आता है जब Stack में कोई भी Data item नहीं होता है वह State underflow कहलाता है।

## Applications of Stack

1. **Reversing a string**—किसी string को Reverse करने के लिए Stack का प्रयोग किया जाता है।

**Example :** Ambition, अगर इस शब्द को उल्टा (Reverse) करना होगा तो पहले इसे Stack में Insert करेंगे इस प्रकार



|   |   |
|---|---|
| n | 7 |
| o | 6 |
| i | 5 |
| t | 4 |
| i | 3 |
| b | 2 |
| m | 1 |
| A | 0 |

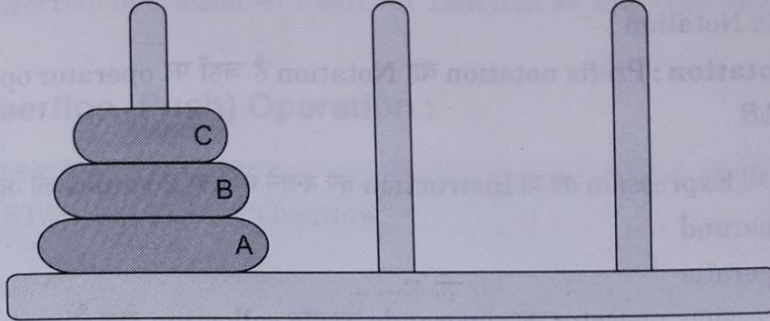
फिर अगर इस Stack में Data को Pop (Delete) करेंगे तो ये शब्द (String) मिलेगा :  
noitibmA

इस प्रकार किसी String/word को reverse किया जा सकता है।

2. **Undo/Redo :** किसी text editor में, किसी Data में किये गये Changes को पुनः वापस पाने के लिए undo का प्रयोग करके step by step उस पुराने text पर जा सकते हैं। और Redo का प्रयोग करके पुनः edit या Change किये हुए Text पर पहुँच सकते हैं। इसका use Excel या word में किया जाता है।
3. किसी Plates या books के Stack को किसी Cupboard पर रखने या लेने के लिए भी Stack की Properties का प्रयोग किया जाता है।
4. **Garage :** किसी Garage में, कोई गाड़ी काफी अन्दर खड़ी है तो इस गाड़ी को निकालने के लिए सारी (उसके पीछे खड़ी गाड़ियों को) गाड़ियों को एक-एक करके बाहर निकालनी होगी। इसके बाद निकाली गयी गाड़ियों को एक-एक करके वापस रखना होगा। यहाँ पर भी Stack की Properties का प्रयोग किया जाएगा।
5. हाथों में चूड़ियाँ पहनने और उनको उतारने के लिए भी Stack की Properties का ही प्रयोग करते हैं।



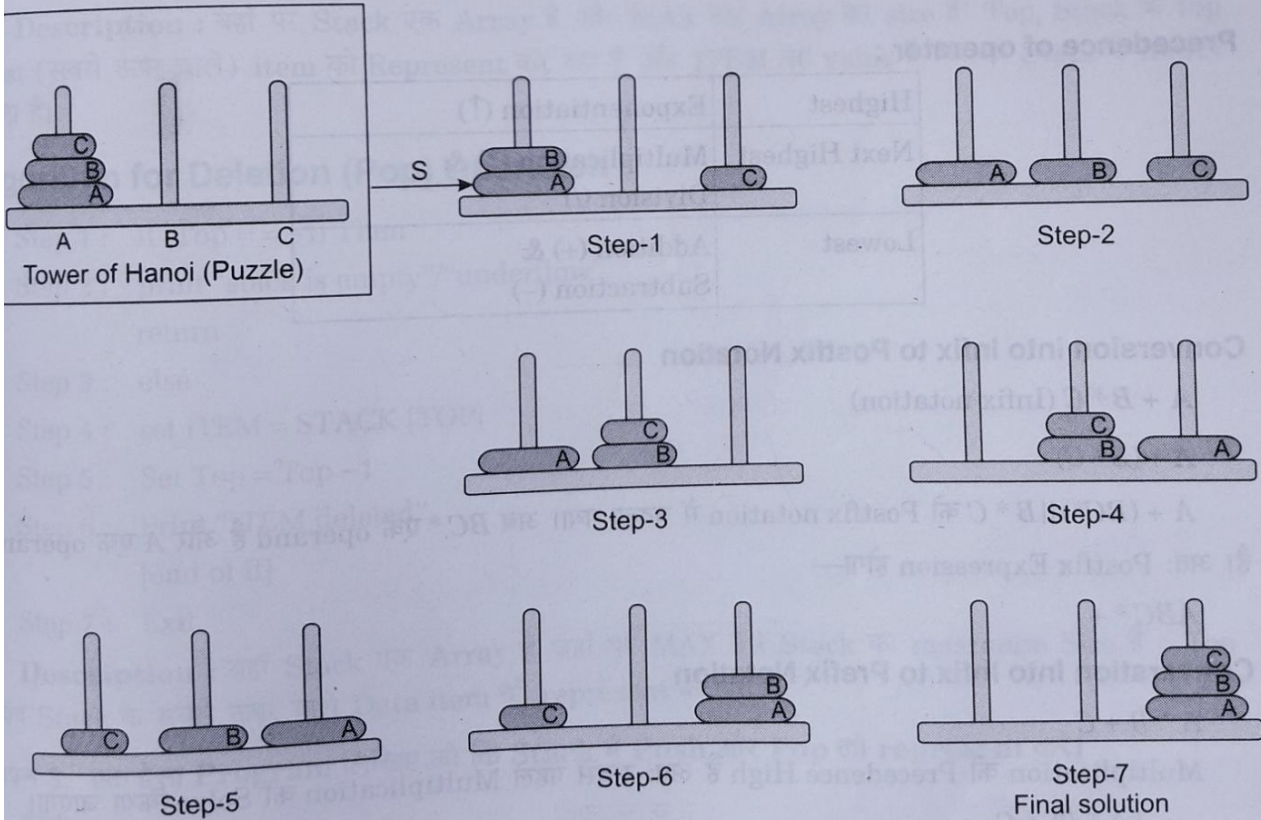
6. **Tower of Hanoi :** यह एक mathematical puzzle (पहेली) है जिसमें तीन (3) towers (पोल) और तीन अलग-अलग size के rings हैं। इस प्रकार



A ring सबसे बड़ा, B उससे छोटा और C सबसे छोटा Ring है। शर्त यह है कि—

1. एक बार में केवल एक ही ring एक Tower से दूसरे Tower पर जाएगा।
2. एक ring के ऊपर दूसरा ring छोटा होना चाहिए।
3. सारे rings अन्त में तीसरे Tower पर होने चाहिए।

इस Puzzle को Stack का use करके solve किया जा सकता है।



7. Stack को Function के बीच Parameter Pass करने के लिए use में किया जाता है।
8. High level Programming Language जैसे Pascal, C आदि में Stack को Recursion के लिए भी use में लेते हैं।



9. Notation में भी Stack को प्रदर्शित किया जाता है। Notation मुख्यतः 3 प्रकार के होते हैं—

1. Prefix Notation

3. Postfix Notation

2. Infix Notation

1. **Prefix Notation** : Prefix notation वह Notation है जहाँ पर operator operand से पहले आता है जैसे कि  $+ AB$

नोट : किसी भी Expression को या Instruction को बनाने के लिए 2 values की आवश्यकता होती है—

1. Operand

2. Operator

Expression, operator और operands का एक collection होता है।

Example  $A + B$

यहाँ पर  $A$  और  $B$  operands हैं और  $+$  operator.

2. **Infix Notation** : Infix notation में operands के बीच में operator होता है। जैसे कि  $A + B$

3. **Postfix Notation** : Postfix Notation वह notation है जहाँ पर operator operand के बाद आता है जैसे  $AB +$

**Precedence of operator :**

|              |                                           |
|--------------|-------------------------------------------|
| Highest      | Exponentiation ( $\uparrow$ )             |
| Next Highest | Multiplication ( $*$ ) & Division ( $/$ ) |
| Lowest       | Addition ( $+$ ) & Subtraction ( $-$ )    |

**Conversion into Infix to Postfix Notation**

$A + B * C$  (Infix notation)

$A + (B * C)$

$A + (BC*)$  को Postfix notation में बदला गया। अब  $BC*$  एक operand है और  $A$  एक operand है। अतः Postfix Expression होगा—

$ABC* +$

**Conversion Into Infix to Prefix Notation**

$A * B + C$

Multiplication की Precedence High है अतः सबसे पहले Multiplication को Solve किया जाएगा।

$(A * B) + C$

$(*AB) + C$

$+ * ABC$



**प्रश्न 2. Stack में Insertion और Deletion के लिए Algorithm लिखिए।**

**उत्तर—**Stack में insertion operation को Push और Deletion को Pop Operation के नाम से भी जाना जाता है।

### Algorithm for Insertion (Push) Operation :

```

Step 1 : if (Top == max - 1) Then
Step 2 : Print "STACK IS FULL"/"Overflow"
          return
Step 3 : else
Step 4 : Set TOP = TOP + 1
Step 5 : Set STACK [TOP] = ITEM
Step 6 : Print "ITEM inserted".
          [End of if]
Step 7 : Exit
  
```

**Description :** यहाँ पर Stack एक Array है और MAX उस Array का size है। Top, Stack के top most (सबसे ऊपर वाले) item को Represent कर रहा है और ITEM वह value है जिसको Stack में Insert करना है।

### Algorithm for Deletion (Pop) Operation :

```

Step 1 : if (Top == -1) Then
Step 2 : print "stack is empty"/"underflow".
          return
Step 3 : else
Step 4 : set ITEM = STACK [TOP]
Step 5 : Set Top = Top - 1
Step 6 : Print "ITEM deleted"
          [end of if]
Step 7 : Exit
  
```

**Description :** यहाँ Stack एक Array है जहाँ पर MAX उस Stack का maximum Size है : Top हमेशा Stack के सबसे ऊपर वाले Data item को represent करेगा।

**प्रश्न 3. एक ऐसा Program लिखिए जो कि Stack में Push और Pop को represent करे।**

**उत्तर—**

```

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

int Push ( ) ;
int Pop ( ) ;
  
```



```

int traverse ( ) ;
int stack [10] ;
int top = - 1 ;
void main ( )
{
    int ch;
    do
    {
        printf("\n 1. Push operation");
        printf("\n 2. Pop operation");
        printf("\n 3. Traverse operation");
        scanf("%d", & ch);
        switch (ch)
        {
            case 1 :
                Push ( ) ;
                break;
            case 2:
                Printf ("\n the deleted element is % d", Pop ());
                break;
            case 3 :
                traverse ( ) ;
                break;
        }
    }
    while (ch != 0);
    getch ( ) ;
}

int push ( )
{
    int item;
    if (top < 9)
    {
        printf ("Enter the element to be inserted :");
        scanf ("%d", & item);
    }
}

```



```
    top += 1;
    stack[top] = item;
    printf("The item inserted at location %d", top);
}
else
{
    printf("The stack is full/overflow condition\n");
}
}

int pop ( )
{
    int item;
    if (top == -1)
    {
        printf("The stack is null and no item can be deleted\n");
    }
    else
    {
        item = stack[top];
        top -= 1;
    }
    return (item);
}

int traverse ( )
{
    int i;
    if (top == -1)
    {
        printf("The stack is null");
    }
    else
    {
        printf("The elements in stack are:");
        for (i = top ; i >= 0; i++)
        {
            printf("%4d", stack[i]);
        }
    }
}
```



```

    }
  }
}

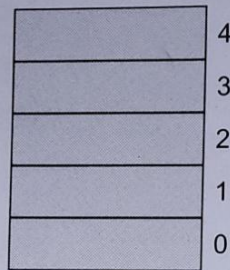
```

**प्रश्न 4. Stack में होने वाले Push और Pop operations को Example के द्वारा समझाइए।**

**उत्तर—Push operation :** Stack में किसी Data item को insert करने के लिए Push operation किया जाता है। जैसे कि—

$A = \{10, 20, 25, 30, 40\}$

**Insert :**



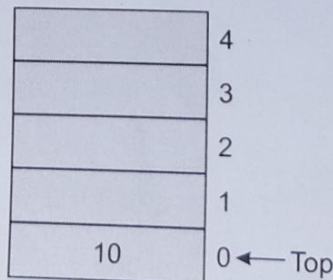
Top = -1

**Step 1 :**

$$\begin{aligned} \text{Top} &= \text{Top} + 1 \\ &= -1 + 1 \end{aligned}$$

$$\text{Top} = 0$$

यहाँ पर Top की Value 0 है।



अर्थात् Stack के 0<sup>th</sup> Position पर पहली Value store/insert होगी।

**Step 2 :**

$$\begin{aligned} \text{Top} &= \text{Top} + 1 \\ &= 0 + 1 \end{aligned}$$

$$\text{Top} = 1$$

अब दूसरा Data item Stack के 1<sup>st</sup> Index पर insert होगा।



|    |         |
|----|---------|
|    | 4       |
|    | 3       |
|    | 2       |
| 20 | 1 ← Top |
| 10 | 0       |

**Step 3 :**

$$\text{Top} = \text{Top} + 1$$

$$= 1 + 1$$

$$\text{Top} = 2$$

Array का 3<sup>rd</sup> Element/Data item Stack के 2<sup>nd</sup> Index पर insert होगा।

|    |         |
|----|---------|
|    | 4       |
|    | 3       |
| 25 | 2 ← Top |
| 20 | 1       |
| 10 | 0       |

**Step 4 :**

$$\text{Top} = \text{Top} + 1$$

$$= 2 + 1$$

$$\text{Top} = 3$$

Array का 4<sup>th</sup> Data item/element, Stack के 3<sup>rd</sup> Index पर insert होगा।

|    |         |
|----|---------|
|    | 4       |
| 30 | 3 ← Top |
| 25 | 2       |
| 20 | 1       |
| 10 | 0       |

**Step 5 :**

$$\text{Top} = \text{Top} + 1$$

$$= 3 + 1$$

$$\text{Top} = 4$$

Array का 5<sup>th</sup> Element/Data item Stack के 4<sup>th</sup> Index पर insert होगा।



|    |         |
|----|---------|
| 40 | 4 ← Top |
| 30 | 3       |
| 25 | 2       |
| 20 | 1       |
| 10 | 0       |

**Step 6 :**

$$\text{Top} = \text{Top} + 1$$

$$= 4 + 1$$

$$\text{Top} = 5$$

यहाँ पर Top की Value 5 है जो कि Stack के maximum size से ज्यादा है। इस situation में कोई भी element Store/Insert नहीं होगा और एक Message "Stack is overflow" Display होगा।

**Pop operation**

किसी Stack में उपस्थित Data items को Remove/Delete करने के लिए Pop operation का प्रयोग किया जाता है। जैसे कि—

|    |         |
|----|---------|
|    | 4       |
|    | 3       |
| 30 | 2 ← Top |
| 20 | 1       |
| 10 | 0       |

यहाँ से Data को इस प्रकार Delete या remove कर सकते हैं।

**Step 1 :**

$$\text{Top} = \text{Top} - 1$$

$$= 2 - 1$$

$$\text{Top} = 1$$

Top की Value 1 है। इस प्रकार top, stack के 1<sup>st</sup> Index पर आ जाएगा, और 2<sup>nd</sup> Index की value delete हो जाएगी।

|    |         |
|----|---------|
|    | 4       |
|    | 3       |
|    | 2 ← Top |
| 20 | 1       |
| 10 | 0       |



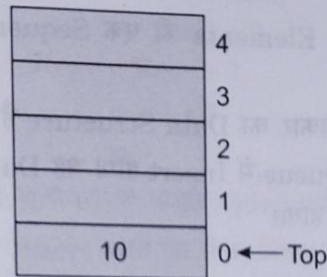
**Step 2 :**

$$\text{Top} = \text{Top} - 1$$

$$= 1 - 1$$

$$\text{Top} = 0$$

अब top की value 0 है अर्थात् अब top, stack के 0<sup>th</sup> Index पर आ जाएगा, तो 1<sup>st</sup> Index की value delete हो जाएगा।

**Step 3 :**

$$\text{Top} = \text{Top} - 1$$

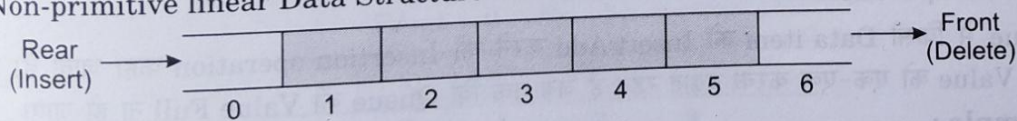
$$= 0 - 1$$

$$\text{Top} = -1$$

Top की value -1 होते ही Stack Empty हो जाएगा जो Stack के Underflow होने की स्थिति को represent करता है।

**प्रश्न 5. Queue क्या होता है? इसके Application को बताइए।**

**उत्तर—Queue :** Queue, data items एक sequential collection होता है जिसमें दोनों छोर (Ends) से operation perform किया जाता है। जिस End से Insertion operation किया जाता है इस end को “rear” और दूसरे end को जिससे Deletion operation किया जाता है इसे “front” कहते हैं। जो Data item, Queue में अन्त (last) में Add/Insert होता है इसे Delete/Remove होने के लिए उसके पहले जितने Data items हैं उनको Delete करना पड़ेगा। इस प्रकार इस sequential collection को First-in-first-out (FIFO) भी कहा जाता है। यह एक Non-primitive linear Data Structure कहलाता है।



**Example 1 :** हमारे जीवन में कभी-न कभी ऐसा situation जरूर आया होगा जहाँ पर हमें किसी Data/Things को लेने के लिए इन्तजार करना पड़ा होगा। जैसे कि—

1. Railway Reservation Counter पर।
2. Movie के Ticket के लिए।
3. शहर के जाम में गाड़ियों को निकालने के लिए। ऐसे कई सारे situations जरूर आये होंगे जहाँ पर हम Queue को जरूर Follow किए होंगे।

**Example 2 :** अगर हम एक Computer Lab की बात करें तो 50 computers, एक printer से जुड़े हैं तो बहुत सारे Students अपने Documents Print करना चाहते हैं। इन सभी



Students की request, Printer में Queue के रूप में store हो जाएगा। जिस order में Printer को request मिला होगा, एक-एक करके उन सभी request/commands को Print करेगा। जो सबसे last में request/command होगा उसे सारे requests Print होने तक इन्तजार करना पड़ेगा।

### Basic Features of Queue

1. STACK की तरह, Queue भी Elements की एक Sequential Collection होता है एक ही तरह के Elements का।
2. Queue, First-in-first-out प्रकार का Data Structure है।
3. नया Data item जो last में Queue में Insert होगा वह Data, सारे Data items के remove होने के बाद (सबसे last में) remove होगा।

### Applications of Queue

Queue, इस Data Structure के नाम से ही पता चलता है कि इसमें कई सारे Data होंगे जो एक Queue (line) में होंगे। इस Queue का किस प्रकार से use किया जाता है वह निम्न है—

1. किसी एक ही Resource (जैसे कि Printer, CPU) को use करने के लिए।
2. Call Centre में, जहाँ पर एक व्यक्ति से बात करने के बाद ही दूसरे व्यक्ति से बात कर सकते हैं।
3. Real time system में जो interruptions आते हैं उनको भी Queue की तरह ही handle किया जाता है।
4. Resources (जैसे कि Printer, Scanner, आदि) के लिए Memory allocation (Memory में जगह देना) भी Queue के रूप में ही होता है।

**प्रश्न 6. Queue पर होने वाले operations का वर्णन करें तथा इनके Algorithm लिखें।**

**उत्तर—**Queue पर मुख्य रूप से दो प्रकार के operations perform किये जाते हैं—

1. Insertion operation (Rear)
2. Deletion Operation (Front)

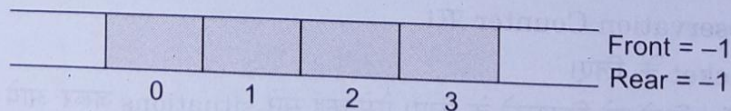
#### 1. Insertion Operation

Queue में किसी Data item को Insert/Add करने को Insertion operation कहा जाता है। इसके लिए Rear की Value को एक-एक करके बढ़ाते रहते हैं जब तक कि Queue की Value Full ना हो जाए।

**Example :**

10, 20, 30, 40

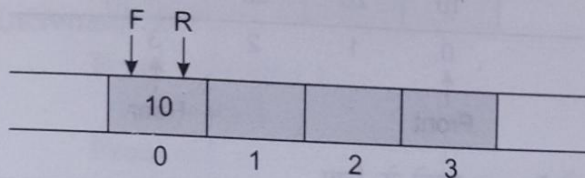
**Step 1 :**



यहाँ पर Queue में कोई Value नहीं है अर्थात् Front और rear की Value -1 है तो पहली Value को Add/Insert करने के लिए Front और Rear दोनों की Value बढ़ेगी।



**Step 2 :**



$$\begin{aligned}\text{Front} &= \text{Front} + 1 \\ &= -1 + 1 = 0\end{aligned}$$

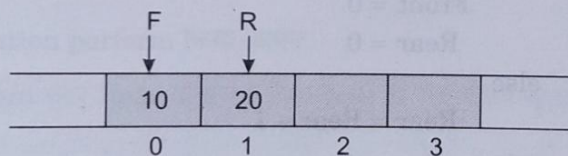
$$\begin{aligned}\text{Rear} &= \text{Rear} + 1 \\ &= -1 + 1 \\ &= 0\end{aligned}$$

जब Front और Rear की Value शून्य (0) है अर्थात् Queue में 0<sup>th</sup> Index पर पहली Value Insert होगी।

**Step 3 :** अब दूसरी Value को Insert करने के लिए केवल Rear value में एक Increment होगा और Front की value नहीं बढ़ेगी।

$$\begin{aligned}\text{Front} &= 0 \\ \text{Rear} &= \text{Rear} + 1 \\ &= 0 + 1 \\ &= 1\end{aligned}$$

दूसरा value index 1<sup>st</sup> पर Insert होगा।

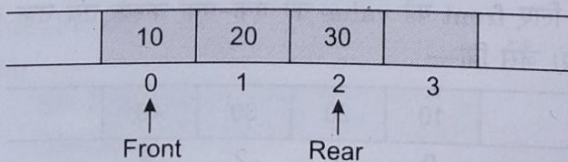


**Step 4 :** तीसरा value Insert करने के लिए फिर

$$\begin{aligned}\text{Rear} &= \text{Rear} + 1 \\ &= 1 + 1\end{aligned}$$

$$\text{Rear} = 2$$

$$\text{Front} = 0$$



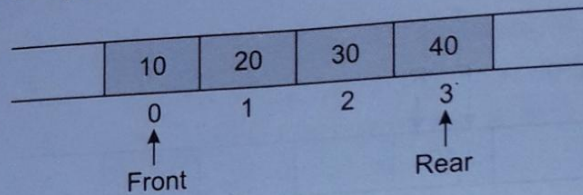
**Step 5 :** अगली Value को Insert करने के लिए

$$\begin{aligned}\text{Rear} &= \text{Rear} + 1 \\ &= 2 + 1\end{aligned}$$

$$\text{Rear} = 3$$

$$\text{Front} = 0$$





**Step 6 :** अगली Value को Insert करने के लिए

$$\begin{aligned} \text{Rear} &= \text{Rear} + 1 \\ &= 3 + 1 = 4 \end{aligned}$$

जो कि Queue की maximum size से ज्यादा है। इस Situation में अब और कोई Data item Insert नहीं हो पायेगा। यह situation overflow कहलाता है।

### Algorithm for Insertion

**Step 1 :** Initialization

set Front = -1

set Rear = -1

**Step 2 :** Repeat Steps 3 to 5 until  $\text{Rear} < \text{max size} - 1$

**Step 3 :** Read item

**Step 4 :** If

Front == -1, then

Front = 0

Rear = 0

else

Rear = Rear + 1

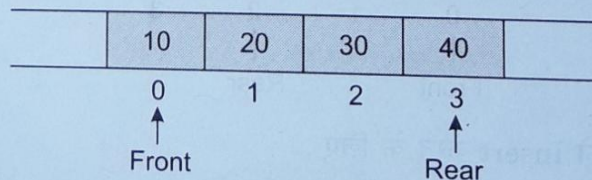
endif

**Step 5 :** Set Queue [Rear] = Item

**Step 6 :** Print, Queue Overflow.

### 2. Deletion operation

Queue में पहले से inserted Data items को Remove या Delete करने के लिए Deletion operation perform किया जाता है। इसके लिए front की value को एक-एक करके तब तक बढ़ाते रहते हैं जब तक कि वह Data item Delete ना हो जाए। जैसे कि—



**Step 1 :** दिए गये Queue से 20 elements को Delete/Remove करना है तो इस प्रकार operation होगा—

Front == 20

10 == 20

सबसे पहले Front की value से, जिस Data item को Delete करना है इससे compare करते हैं कि front का Data वही Data तो नहीं है जिसे Delete करना है।



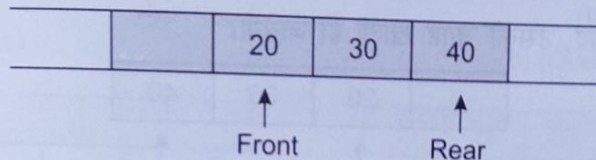
Front != 20

अब Front की value को Increment करेंगे

Front = Front + 1

= 0 + 1

Front = 1



Step 2 :

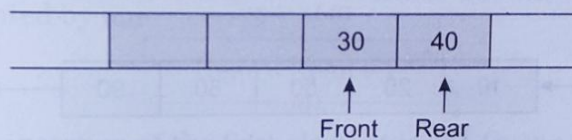
Front == 20

यहाँ पर Front की value 20 है। अब Front की value को बार Increment करने पर यह value Delete हो जाएगा।

Front = Front + 1

= 1 + 1

Front = 2



इस प्रकार Deletion operation perform किया जाएगा।

नोट : 1. Queue में Front और Rear दोनों एक Index पर हों उस समय Queue में एक Data item होगा।

2. Deletion के समय दोनों (Front और Rear) एक Index पर हों तो वह Queue का last Data item है इसे Delete करने के लिए Front और Rear दोनों की value -1 करना होगा।

### Algorithm for Deletion

Step 1 : Repeat Step 2 to 4 until front >= 0

Step 2 : Set item = queue [Front]

Step 3 : if Front == Rear

Set Front = -1

Set Rear = -1

else

Front = Front + 1

Step 4 : Print, Deleted item is, Item.

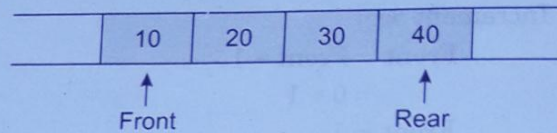
Step 5 : Print, "Queue is empty".

प्रश्न 7. Circular Queue क्या होता है? संक्षिप्त में वर्णन कीजिए।

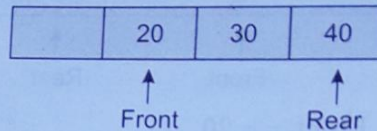
उत्तर—Circular Queue : Queue में Data को Delete करने के बाद जो Queue में जगह बच/खाली हो जाता है इसे पुनः तब तक भरा नहीं जा सकता जब तक कि सारे Data Delete न हों।



जैसे कि—

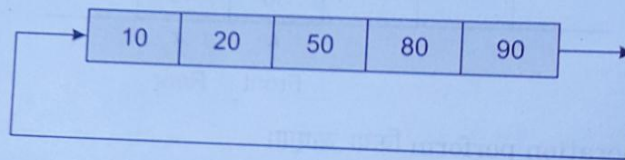


10 को Delete करने के बाद, उसकी जगह खाली हो जायेगी।



इस खाली जगह को पुनः नहीं भरा जा सकता। Queue के इस Drawback को Circular Queue के द्वारा खल किया जा सकता है।

Circular Queue को हम ring - buffer भी कहते हैं। Circular Queue में जो अंतिम नोड होता है वह सबसे पहले node से जुड़ा हुआ रहता है जिससे कि circle का निर्माण होता है। यह FIFO के सिद्धान्त पर कार्य करता है। Circular Queue में item को rear end से Add किया जाता है तथा item को front end से remove किया जाता है।



## Queue

Queue is a non-linear data structure, somewhat similar to stacks. Unlike stack, a queue is open at both ends. One end is always used to insert data (enqueue : the process of adding an element to the collection, the element is added from the rear side) and the other used to remove data (dequeue : the process of removing the first element that was added, the element is removed from the front side). Queue follows first-in-first-out, methodology. i.e., the data item stored first will be accessed first. It can be implemented by using both array and linked list.

A real world example of queue can be single lane one-way road where the vehicle enters first, exits first, more real world examples can be seen as queue at the ticket windows and bus-stops.

It means the customers are serviced in the order in which they arrive at the service center First come first service (FCFS) types of service).

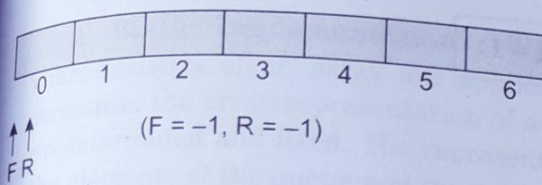
## Operation on Queue

### 1. Insertion

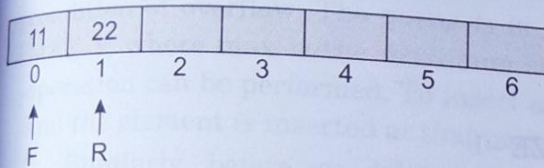
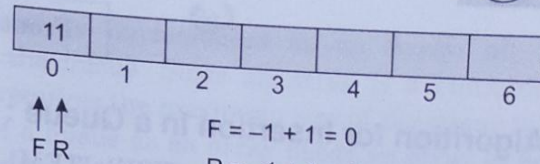
### 2. Deletion

**1. Insertion :** Adding a new element in a Queue is called Insertion. Its process is in this way :

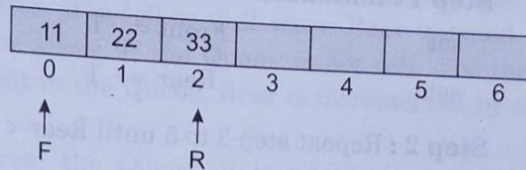




(a) (Empty Queue)

(c) ( $F = 0, (R = R + 1 = 0 + 1)$   
( $F = 0, R = 1$ )

(b) (Queue with one element)

(d) ( $F = 0, R = R + 1 = 2$ )

It is clear from the above figure that whenever we insert an element in the Queue, the value of Rear is incremented by one

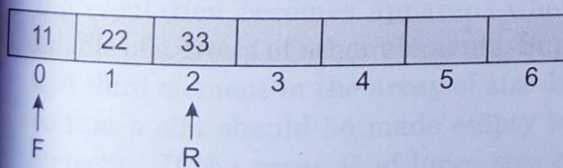
$$\text{Rear} = \text{Rear} + 1$$

Note that during the insertion of the first element in the Queue we always increment the front by one

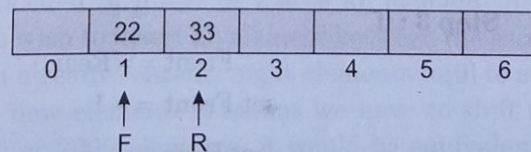
$$\text{Front} = \text{Front} + 1$$

After this the front will not be changed during the entire addition operation.

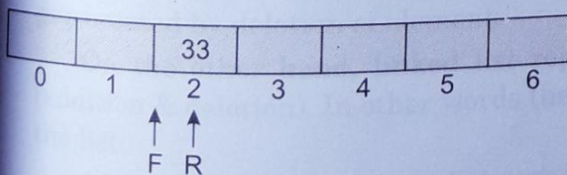
**2. Deletion :** Removing of an element from Queue is called deletion. The following figures show the deletion :



(a) (Queue with 3 elements)



(b) (Queue with 2 elements)



(c) (Queue with 1 element)

This is clear from the above figure that whenever an element is removed or deleted from the Queue, the value of front is incremented by one.



Front = Front + 1

### Algorithm for Insertion in a Queue :

Qinsert [Queue [max SIZE], ITEM]

This algorithm inserts an item at the rear of the Queue.

#### Step 1 : Initialization

set Front = - 1

set Rear = - 1

**Step 2 :** Repeat step 3 to 5 until  $\text{Rear} < \text{max SIZE} - 1$

**Step 3 :** Read item

**Step 4 :** if Front == -1, then

Front = 0

Rear = 0

else

Rear = Rear + 1

end if

**Step 5 :** set Queue [Rear] = ITEM

**Step 6 :** print, Queue overflow.

### Algorithm for Deletion from a Queue

QDELETE [Queue [MAX SIZE], ITEM]

**Step 1 :** Repeat Step 2 to 4 until  $\text{front} > = 0$

**Step 2 :** set item = Queue [Front]

**Step 3 :** if

Front == Rear

set Front = - 1

set Rear = - 1

else

Front = Front + 1

**Step 4 :** print, no deleted is, ITEM

**Step 5 :** print. Queue is Empty.

### Implementation of Queue

Queue can be represented in two ways :

1. Static Implementation (using Array)
2. Dynamic Implementation (Using Pointers)



**1. Static Implementation :** When Queue is implemented as an Array, all the characteristics of an array are applicable to the queue. Since an array is a static data structure, the array representation of a queue requires the maximum size of the queue to be predetermined and fixed. The representation of a queue as an array needs an array to hold the elements of the queue and two variables, rear and front, to keep track of the rear and the front ends of the queue respectively. Initially the value of rear and front is set to -1 to indicate an empty queue. Before we insert a new element in the queue, it is necessary to test the condition of overflow. The queue is in a condition of overflow (full) when Rear is equal to MAX-1, where max is the maximum size of the array. If the Queue is not full. The insert operation can be performed. To insert an element in the Queue, Rear is incremented by one, and the element is inserted at that position.

Similarly, before we delete an element from the Queue, it is necessary to test the condition of underflow. The Queue is in the condition of underflow (empty) when the value of front is -1. If the Queue is not empty, delete operation can be performed. To delete an element from the Queue, the element referred by the front is assigned to a local variable and then front is incremented by one.

The total number of elements in a Queue at a given point of time can be calculated from the value of Rear and front as given here.

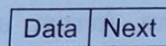
$$\text{Number of Elements} = \boxed{\text{Rear} - \text{front} + 1}$$

**2. Implementation of Queue Using Pointers :** Implementing Queues (and stacks) using Pointers, the main disadvantage is that a node in a linked list representation (using Pointers) occupies more memory space than a corresponding element in the array representation. Since there are at least two fields in each node one the data field and the other to store the address of next node, whereas the linked list representing only data field is there. But note that the memory space occupied by a node in a linked representation is not exactly twice the space used by an element of array representation. Moreover a linked representation makes an effective utilization of memory. The advantage of linked list representation becomes apparent when it is needed to insert or delete an element in the middle of a group of other elements. Suppose we wish to insert an element between the second and third element in the array of size 10, which already contains eight elements (a[0] to a[7]) so that a slot should be made empty to insert new element. It means we have to shift five elements. If the array is of large size say 1000 or 2000 elements, it would be equivalent of doing a bulk of work and gaining no special results. Similarly when an element in between is deleted it requires all the elements beyond the deleted element to be shifted, so as to fill the gap created by deletion of element.

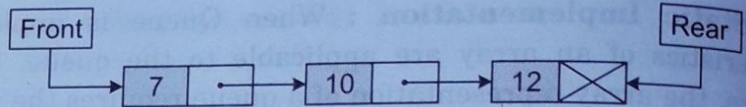
On the other hand, linked list representation allow us to stress on our primary aim (addition & deletion). In other words the amount of work required is independent of the size of the list.

Addition of a new node in between a queue in a linked presentation requires creating a new node, inserting it in the required position by adjusting two pointers. An individual node and queue in linked representation is shown in the figure.





(a) A node



(b) A queue

## Implementation of Queue

Stack की तरह, Queue को भी दो तरह से Implement किया जाता है—

1. Static implementation (using Array)
2. Dynamic implementation (using Pointer)

### 1. Static implementation

अगर Queue को Array के द्वारा Implement किया जा रहा है तो यह Declaration time पर ही Array का size fix हो जाता है। इस तरह Array का Starting Position front और last Position rear कहलायेगा।

$$\text{front} - \text{rear} + 1$$

यह Relation बताता है कि Array में कुल कितने elements हैं।

**Note :** यदि  $\text{rear} < \text{front}$  तो Queue में एक भी Element नहीं है अर्थात् Queue खाली है।

// Program to implement linear queue using Array.

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

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

```
int Queue [5];
```

```
long front, rear;
```

```
void display ( );
```

```
void main ( )
```

```
{
```

```
int choice, info;
```

```
clrscr ( );
```

```
while (1)
```

```
{
```

```
clrscr ( );
```

```
printf("\n 1. Insert an element in Queue");
```

```
printf("\n 2. Delete an Element from Queue");
```

```
printf("\n 3. Display the Queue");
```

```
printf("\n 4. Exit");
```

```
printf("your choice");
```

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

```
switch (Choice)
```



```
{
    case 1 :
        if (rear < 4)
        {
            printf("enter the number");
            scanf("%d", & info);
            if (front == -1)
            {
                front = 0;
                rear = 0;
            }
            else
                rear = rear + 1;
            Queue [rear] = info;
        }
        else
            printf ("Queue is full");
            getch ( ) ;
            break;
    case 2 :
        int info;
        if (front != - 1)
        {
            info = Queue [front];
            if (front == rear)
            {
                front = -1;
                rear = -1;
            }
            else
                front = front + 1;
            printf ("no deleted is = %d", info);
        }
        else
            printf("Queue is Empty");
}
```



```

        getch ( ) ;
        break ;

    case 3 :
        display ( ) ;
        getch ( ) ;
        break ;

    case 4 :
        exit ( ) ;
        break ;

    default;
        printf ("you entered wrong choice");
        getch ( ) ;
        break ;
    }
}
}

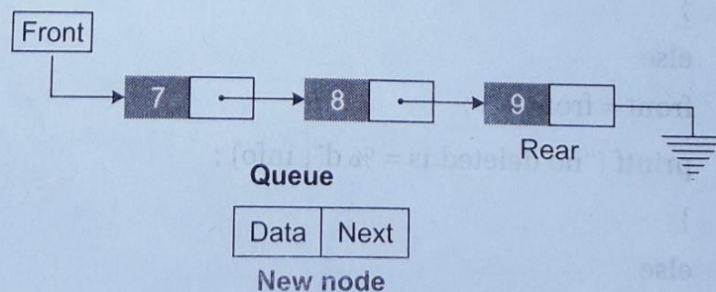
```

## 2. Dynamic implementation (using linked list)

Stack और Queue का implementation Pointer के द्वारा करने पर memory ज्यादा occupied (लगाता) होता है क्योंकि यहाँ पर दो Fields होते हैं पहला Data field जो Data को Store करने के लिए use होता है और दूसरा Address/link field जो कि दूसरे node को link करने के लिए use होता है। इसके बावजूद Memory का अच्छा utilization, link list के द्वारा होता है। इसके द्वारा linked list में insertion और Deletion कहीं पर भी किया जा सकता है।

माना कि एक Array जिसका size 10 है इसमें 2<sup>nd</sup> और 3<sup>rd</sup> elements के बीच में एक Array Element को Store/Insert करना चाहते हैं परन्तु Array में 8 elements a[0] to a[7] पहले से ही हैं। Insertion करने के लिए elements a[2] से a[7] तक सभी को एक Place Shift होना पड़ेगा। इसका मतलब पाँच (5) elements को shift करना होगा और खाली जगह पर नया Element insert हो जायेगा पर elements की size ज्यादा हो 1000 या 2000 तो यह एक complicated task हो जायेगा। इसी तरह अगर बीच से किसी Element को delete कर दिया जाए तो सारे elements को, Delete हुए Element की ओर Shift करना पड़ेगा।

Linked list में new element को Insert करने के लिए एक node create करना पड़ेगा।





// This Program represents the working of a Queue when  
// implemented by Pointers.

```
#include <stdio.h>
#include <conio.h>
struct Queue
{
```

```
    int no;
    struct Queue *next;
```

```
};
*Start = NULL;
```

```
void add ( );
```

```
int del ( );
```

```
void traverse ( );
```

```
void main ( )
```

```
{
```

```
    int ch;
```

```
    char choice ;
```

```
    do
```

```
    {
```

```
        int ch ;
```

```
        char choice ;
```

```
        do
```

```
        {
```

```
            clrscr ( );
```

```
            printf ("\n 1. add");
```

```
            printf ("\n 2. delete");
```

```
            printf ("\n 3. trarerse");
```

```
            printf ("\n 4. Exit");
```

```
            printf ("Enter your choice");
```

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

```
            switch (ch)
```

```
            {
```

```
                case 1:
```

```
                    add ( );
```

```
                    break ;
```

```
                case 2 :
```



```
        printf ("the Deleted Element is \n%d", del ( ));
        break;
    case 3 :
        traverse ( );
        break;
    case 4 :
        return;
    default :
        printf ("worng choice");
    }
    fflush (stdin);
    scanf ("%c", & choice);
}
while (choice != 4);
}

void add ( )
{
    struct queue *P, *temp;
    temp = Start;
    P = (struct Queue*) malloc (Size of (struct Queue));
    printf ("Enter the data");
    scanf ("%d", & p -> no);
    P - next = NULL;
    if (start == NULL)
    {
        Start = P;
    }
    else
    {
        while (temp -> next != NULL)
        {
            temp = temp -> next ;
        }
        temp -> next = P;
    }
}
```



```

int del ( )
{
    struct Queue *temp;
    int value;
    if (Start == NULL)
    {
        printf ("Queue is Empty");
        getch ( );
        return (0);
    }
    else
    {
        temp = start ;
        value = temp -> no;
        start = start -> next ;
        free (temp);
    }
    return (value);
}

void traverse ( )
{
    struct Queue *temp;
    temp = start;
    while (temp -> next != NULL)
    {
        printf ("no = %d", temp -> no);
        temp = temp -> next ;
    }
    printf ("no = %d", temp -> no);
    getch ( );
}

```

### Applications of Stack Data Structure

The linear data structure stack can be used in the following situations.

1. It can be used to process function calls.



2. Implementing recursive functions in high level languages.
3. Converting and evaluating expressions.

### Function calls :

A stack is useful for the compiler/operating system to store local variables used inside a function block, so that they can be discarded once the control comes out of the function block.

### Recursive functions :

The stack is very much useful while implementing recursive functions. the return values and addresses of the function will be pushed into the stack and the lastly invoked function will first return the value by popping the stack.

### Representation of Expressions :

In general there are 3 kinds of expressions available depending on the placement of the operators & operands.

1. **Infix expression** : It is the general notation used for representing expressions.  
 "In this expression the operator is fixed in between the operands".  
**Ex** :  $a + bc$
2. **Postfix expression** : (Reverse polish notation)  
 "In this expression the operator is placed after the operands".  
**Ex** :  $abc +$
3. **Prefix expression** : (Polish notation)  
 "In this expression the operators are followed by operands i.e. the operators are fixed before the operands"  
**Ex** :  $+ abc$

All the infix expressions will be converted into postfix notation with the help of stack in any program.

The stack will be useful in evaluating the postfix expressions also.

### Algorithm for evaluating postfix expression using stacks :

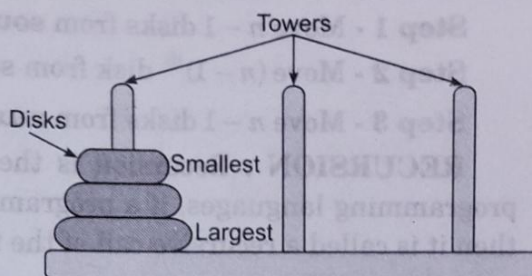
- Step 1 : start
- Step 2 : for (each character ch in the postfix expression)
- Step 3 : If operand is found push it into the stack
- Step 4 : else
- Step 5 : If operator is found then pop the stack 2 times  
 $OP2 = \text{pop}() \quad OP1 = \text{pop}()$
- Step 6 : Perform the specified operation as  $\text{result} = OP1 \text{ operator } OP2$
- Step 7 : Push the intermediate result back into the stack
- Step 8 : Repeat the above steps until the end of the expression
- Step 9 : pop the stack to obtain the final result
- Step 10 : stop



### Tower of Hanoi :

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one ring is as depicted :

These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.



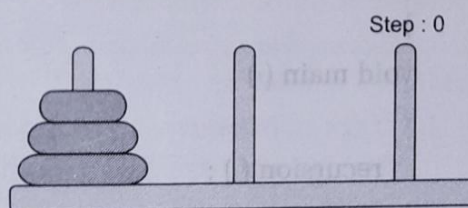
### Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are :

- ❖ Only one disk can be moved among the towers at any given time.
- ❖ Only the “top” disk can be removed.
- ❖ No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.

Tower of Hanoi puzzle with  $n$  disks can be solved in minimum  $2^n - 1$  steps. This presentation shows that a puzzle with 3 disks has taken  $2^3 - 1 = 7$  steps.



### Algorithm

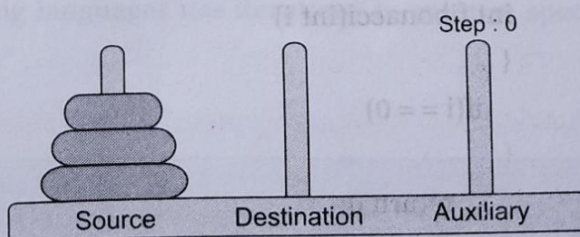
To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say  $\rightarrow 1$  or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks :

- ❖ First, we move the smaller (top) disk to aux peg.
- ❖ Then, we move the larger (bottom) disk to destination peg.
- ❖ And finally, we move the smaller disk from aux to destination peg.

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk ( $n^{\text{th}}$  disk) is in one part and all other  $(n - 1)$  disks are in the second part.

Our ultimate aim is to move disk  $n$  from source to destination and then put all other  $(n - 1)$  disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.





The steps to follow are :

**Step 1** - Move  $n - 1$  disks from **source** to **aux**

**Step 2** - Move  $(n - 1)^{\text{th}}$  disk from **source** to **dest**

**Step 3** - Move  $n - 1$  disks from **aux** to **dest**

**RECURSION** : Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion ( )
{
    recursion ( );/*function calls itself*/
}

void main (·)
{
    recursion ( ) ;
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems such as calculating the factorial of a number, generating Fibonacci series, etc.

The following example generates the Fibonacci series for a given number using a recursive function :

```
#include <stdio.h>
int fibonacci(int i)
{
    if(i == 0)
    {
        return 0;
    }
    if(i == 1)
    {
        return 1;
    }
    return fibonacci (i-1) + fibonacci (i-2);
}

void main ( )
{
}
```



```

int i;
for (i = 0; i < 10; i++)
{
    printf("%d\t\n", fibonacci(i));
}

```

When the above code is compiled and executed, it produces the following result :

```

0
1
1
2
3
5
8
13
21
34

```

### Iteration

With respect to computing, **iteration** is the process of going through a set of operations that deal with computer code. For example, in a computer program, one form of iteration is a loop. A loop repeats code until a certain condition is met. Each time the computer runs through a loop, it is referred to as an iteration.

Many computer programs and programming languages use iterations to perform specific tasks, solve problems, and present solutions.

### Comparison Chart

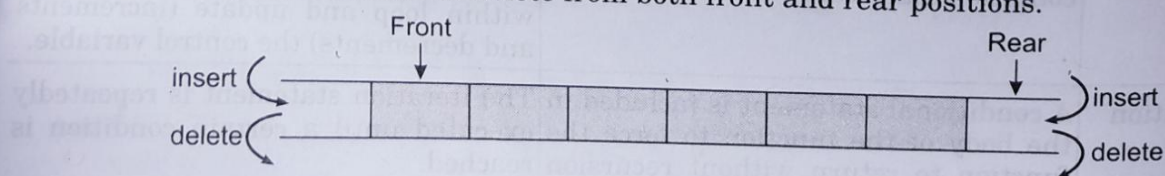
| Basis For Comparison | Recursion                                                                                                                              | Iteration                                                                                                                                     |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| Basic                | The statement in a body of function calls the function itself.                                                                         | Allows the set of instructions to be repeatedly executed.                                                                                     |
| Format               | In recursive function, only termination condition (base case) is specified.                                                            | Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable. |
| Termination          | A conditional statement is included in the body of the function to force the function to return without recursion call being executed. | The iteration statement is repeatedly executed until a certain condition is reached.                                                          |



|                     |                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                        |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Condition           | If the function does not converge to some condition called (base case), it leads to infinite recursion.                                                                                                                | If the control condition in the iteration statement never become false, it leads to infinite iteration.                                                                                                                                                                                |
| Infinite Repetition | Infinite recursion can crash the system.                                                                                                                                                                               | Infinite loop uses CPU cycles repeatedly.                                                                                                                                                                                                                                              |
| Applied             | Recursion is always applied to functions.                                                                                                                                                                              | Iteration is applied to iteration statements or "loop".                                                                                                                                                                                                                                |
| Stack               | The stack is used to store the set of new local variables and parameters each time the function is called.                                                                                                             | Does not use stack.                                                                                                                                                                                                                                                                    |
| Overhead            | Recursion posses the overhead of repeated function calls.                                                                                                                                                              | No overhead of repeated function call.                                                                                                                                                                                                                                                 |
| Speed               | Slow in execution.                                                                                                                                                                                                     | Fast in execution.                                                                                                                                                                                                                                                                     |
| Size of Code        | Recursion reduces the size of the code.                                                                                                                                                                                | Iteration makes the code longer.                                                                                                                                                                                                                                                       |
| Code/logic          | <pre> int factorial(int num) {     int answer;     if (num == 1)     {         return 1;     }     else     {         answer = factorial(num-1) * num;         //recursive calling     }     return (answer); } </pre> | <pre> int factorial(int num) {     int answer = 1; /*needs     initialization because it may     contain a garbage value     before its initialization */     for(int t = 1; t &gt; num; t++)     //iteration     {         answer = answer * (t);     }     return (answer); } </pre> |

## Double Ended Queue (Deque)

Double Ended Queue is also a Dequeue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Double Ended Queue can be represented in TWO ways, those are as follows :

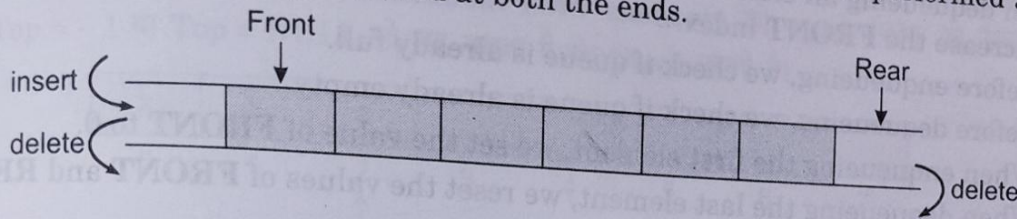
1. Input Restricted Double Ended Queue.



## 2. Output Restricted Double Ended Queue

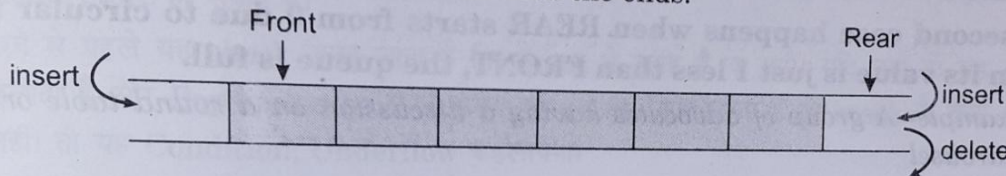
### Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



### Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



## Applications of Dequeue

Since Dequeue supports both stack and queue operations, it can be used as both. The Dequeue data structure supports clockwise and anticlockwise rotations in  $O(1)$  time which can be useful in certain applications.

**Circular queue :** The **queue** is considered as a **circular** when the position 0 and  $\text{MAX}-1$  are adjacent to each other. This means any point before front is also after rear. Circular queue follows FIFO principle.

Circular queue avoids the wastage of space in a regular queue implementation using arrays.

As you can see in the above image, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.

The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

## How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

i.e.,

if  $\text{REAR} + 1 == 5$  (overflow!),  $\text{REAR} = (\text{REAR} + 1) \% 5 = 0$  (start of queue)

### Queue operations work as follows :

- ❖ Two pointers called **FRONT** and **REAR** are used to keep track of the first and last elements in the queue.



- ❖ When initializing the queue, we set the value of **FRONT** and **REAR** to -1.
- ❖ On enqueueing an element, we circularly increase the value of **REAR** index and place the new element in the position pointed to by **REAR**.
- ❖ On dequeueing an element, we return the value pointed to by **FRONT** and circularly increase the **FRONT** index.
- ❖ Before enqueueing, we check if queue is already full.
- ❖ Before dequeueing, we check if queue is already empty.
- ❖ When enqueueing the first element, we set the value of **FRONT** to 0.
- ❖ When dequeueing the last element, we reset the values of **FRONT** and **REAR** to -1.

However, the check for full queue has a new additional case :

- ❖ Case 1 :  $\text{FRONT} = 0 \ \&\& \ \text{REAR} == \text{SIZE} - 1$
- ❖ Case 2 :  $\text{FRONT} = \text{REAR} + 1$

The second case happens when **REAR** starts from 0 due to circular increment and when its value is just 1 less than **FRONT**, the queue is full.

For example- A group of advocates having a discussion on a round table or an airport baggage carousel

### Applications

- ❖ In re-buffering problem, circular queue joins the front and rear part which makes it easier to solve.
- ❖ There are queues of processes in operating system that are waiting for a particular function to occur.
- ❖ You could even use a circular queue to simulate and analyse real world queues.

### Static Implementation

Stack का Static implementation करने के लिए Array का use किया जाता है। Static implementation एक बहुत ही सरल तरीका है पर यह flexible (changeable) नहीं है। जैसे किसी stack की size, Program को Design करते समय declare कर दिया गया तो उसकी Size को change नहीं किया जा सकता। Memory को सही तरीके से use करने के लिए static initialization effective नहीं होता है। किसी stack पर operation करने से पहले, Array की size Declare करना पड़ता है। अगर Declared size से कम elements हों तो फिर Memory (unused) बेकार (waste) हो जाता है, जबकि अगर declared size से ज्यादा element हों तो इस Array की size बाद में बढ़ नहीं सकती जिससे कि इसमें कोई नया element store हो सके।

### Dynamic implementation

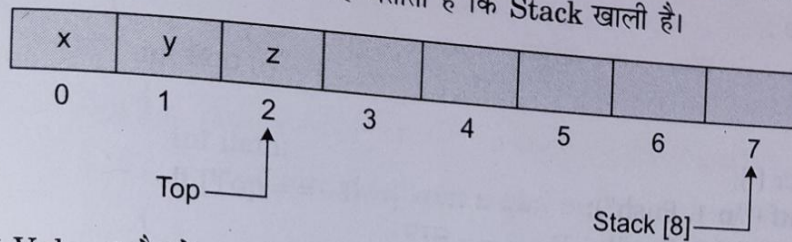
Dynamic implementation is also called linked list representation and uses Pointers to implement the stack type of data structure.

(Dynamic implementation को linked list implementation के नाम से भी जाना जाता है। इस तरह के implementation को represent करने के लिए Pointer का प्रयोग किया जाता है।)



## Array Representation of Stack

Stack को computer में कई प्रकार से represent कर सकते हैं जिसमें Array एक है। इसमें एक Pointer variable Top use किया जाता है जो कि stack के top element को represent करता है और एक variable MAXSTK जो बताता है कि stack में maximum कितने elements को store किया जा सकता है। एक Condition  $\text{Top} = -1$  या  $\text{Top} = \text{NULL}$  जो यह बताता है कि Stack खाली है।



यहाँ पर Top की Value 2 है जो यह बताता है कि इस Stack में तीन (3) element (x, y और z) है तथा और पांच (5) elements store हो सकते हैं।

Push करने से पहले यह check किया जाता है कि Stack में जगह है या नहीं। यदि नहीं तो यह condition, overflow कहलाता है और Pop operation करते समय यह check किया जाता है कि stack में कोई element है कि नहीं यदि नहीं। तो यह Condition, Underflow कहलायेगा।

Stack में Push और Pop operation करने के लिए अलग-अलग Procedures हैं जो निम्न हैं :

### Procedure 1 : Push (STACK, TOP, MAXSTK, ITEM)

इस procedure के द्वारा किसी भी ITEM को Stack में insert किया जा सकता है।

1. [STACK already filled?]

If  $\text{TOP} = \text{MAXSTK}$ , then Print "overflow" and Return.

2. Set  $\text{TOP} = \text{TOP} + 1$  [Increases Top by 1]
3. Set  $\text{STACK}[\text{TOP}] = \text{ITEM}$  [Insert ITEM in new TOP position]
4. Return.

### Procedure 2 : POP (STACK, TOP, ITEM)

इस Procedure के द्वारा Stack के Top से किसी भी item को delete किया जा सकता है।

1. [Stack has an item to be removed?]

If  $\text{TOP} = -1$ , then Print Underflow and Return

2. Set  $\text{ITEM} = \text{STACK}[\text{TOP}]$ . [Assign Top element to ITEM]
3. Set  $\text{TOP} = \text{TOP} - 1$  [Decreases Top by 1.]

**Program :** एक ऐसा Program लिखिए जो stack के operations को represent करे, Array का use करके।

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAXSIZE 10
void Push ( );
int Pop ( );
```



```

void traverse ( );
int stack [MAXSIZE];
int top = -1;
void main ( )
{
    int choice;
    char ch;
    do
    {
        clrscr ( );
        printf ("\n 1. Push");
        printf ("\n 2. Pop");
        printf ("\n 3. Traverse");
        printf ("\n Enter Your Choice");
        scanf ("%d", & choice);
        switch (choice)
        {
            case 1: Push ( );
                    break;
            case 2 :
                    printf ("\n the deleted element is %d", Pop ( ));
                    break;
            case 3 :
                    Traverse ( );
                    break;
            default :
                    printf ("\n you entered wrong choice");
        }
        printf ("\n Do You wish to continue (Y/N)");
        fflush (stdin);
        scanf ("%c", & ch);
    }
    while (ch = 'Y' || ch = 'y');
}

void Push ( )
{
    int item;
    if (top == MAXSIZE -1)
    {
        printf ("\n the stack is full");
        getch ( );
    }
    else

```



```

{
    printf ("enter the element to be inserted");
    scanf ("%d", & item);
    top = top + 1;
    stack [TOP] = item;
}
}

int Pop ( )
{
    int item;
    if (Top == -1)
    {
        printf ("The stack is empty");
        getch ( );
    }
    else
    {
        item = stack [top];
        top = top - 1;
    }
    return (item);
}

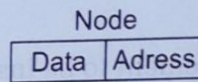
void traverse ( )
{
    int i;
    if (top == -1)
    {
        printf ("The stack is empty");
        getch ( );
    }
    else
    {
        for (i = Top; i >= 0; i --)
        {
            printf ("traverse the element");
            printf ("\n%d", stack [i]);
        }
    }
}

```

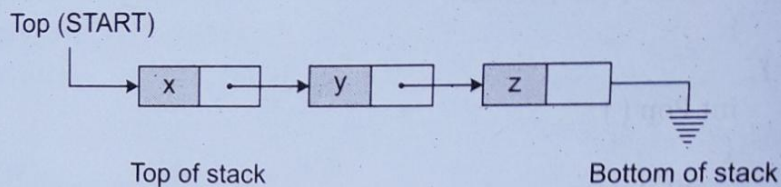
### Linked list Representation of Stack

Stack को linked list के द्वारा represent करने के लिए singly linked list का प्रयोग किया जाता है।



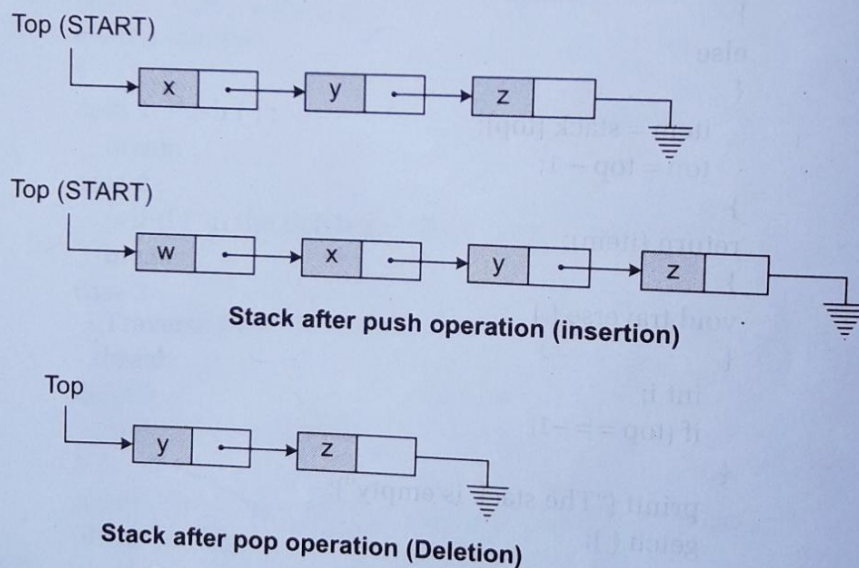


जिसमें Data field, stack के item को store करेंगे और Address field, stack के दूसरे Data को पहले node से link करेगा।



यहाँ पर linked list का Start Pointer, Stack के Top Pointer की तरह use होगा और stack के last item का NULL Pointer bottom item को represent करेगा।

Push operation Perform करने के लिए item को front या start में insert करेंगे और start से ही Data item Delete भी करेंगे।



// यह Program, Stack के operation को linked list के द्वारा किस प्रकार किया जायेगा, बताएगा :

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

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

```
struct stack
```

```
{
```

```
    int no;
```

```
    struct stack * next;
```

```
}
```

```
* top = NULL;
```

```
typedef struct stack st;
```

```
void push ( );
```



```
int Pop ( );
void display ( );
void main ( )
{
    char ch;
    int choice, item;
    do
    {
        clrscr ( );
        printf ("\n 1. Push");
        printf ("\n 2. Pop");
        printf ("\n 3. display");
        printf ("\n 4. exit");
        printf ("\n enter your choice");
        scanf ("%d", & choice);
        switch (choice)
        {
            case 1 :
                Push ( );
                break;
            case 2 :
                item = pop ( );
                printf ("The deleted element is %d", item);
                break;
            case 3 :
                display ( );
                break;
            default :
                printf ("\n wrong choice");
        };
        printf ("\n do you want to continue (Y/n)");
        fflush (stdin);
        scanf ("%c", & ch);
    }
    while ((ch = 'Y') && (ch = 'y'));
}

void Push ( )
{
    st * P;
    node = (st *) malloc (size of (st));
    printf ("\n enter the number");
```



```

scanf ("%d", &P -> no);
P -> next = top;
top = node;
}

int Pop ( )
{
    st * P :
    P = start;
    if (top == NULL)
    {
        printf ("stack is already empty");
        getch ( ) ;
    }
    else
    {
        top = top -> next :
        free (P);
    }
    return (P -> no);
}

void display ( )
{
    st * P :
    temp = top;
    while (P -> next != NULL)
    {
        printf ("\n no = %d", P -> no);
        P = P -> next ;
    }
    printf ("\n no = %d", P -> no);
}

```



# Sorting Algorithms

## Sorting

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios :

- ❖ **Telephone Directory** : The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- ❖ **Dictionary** : The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

## Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques :

### Insertion sort

In this method, sorting is done by inserting elements into an existing sorted list. Initially, the sorted list has only one element. Other elements are gradually added into the list in the proper position.

### Merge sort

In this method, the elements are divided into partitions until each partition has sorted elements. Then, these partitions are merged and the elements are properly positioned to get a fully sorted list.

### Quick sort

In this method, an element called pivot is identified and that element is fixed in its place by moving all the elements less than that to its left and all the elements greater than that to its right.

### Radix sort

In this method, sorting is done based on the place values of the number. In this scheme, sorting is done on the less-significant digits first. When all the numbers are sorted on a more



significant digit, numbers that have the same digit in that position but different digits in a less-significant position are already sorted on the less-significant position.

### Heap sort

In this method, the file to be sorted is interpreted as a binary tree. Array, which is a sequential representation of binary tree, is used to implement the heap sort.

The basic premise behind sorting an array is that its elements start out in some random order and need to be arranged from lowest to highest.

It is easy to see that the list

1, 5, 6, 19, 23, 45, 67, 98, 124, 401

is sorted, whereas the list

4, 1, 90, 34, 100, 45, 23, 82, 11, 0, 600, 345

is not. The property that makes the second one "not sorted" is that there are adjacent elements that are out of order. The first item is greater than the second instead of less, and likewise the third is greater than the fourth and so on. Once this observation is made, it is not very hard to devise a sort that proceeds by examining adjacent elements to see if they are in order, and swapping them if they are not.

### Selection sort

In this technique, the first element is selected and compared with all other elements. If any other element is less than the first element swapping should take place. By the end of this comparison, the least element most top position in the array. This is known as pass 1. In pass II, the second element is selected and compared with all other elements. Swapping takes place if any other element is less than selected element. This process/continues until array is sorted.

The no. of passes in array compares to size of array-1.

### Bubble sort

This technique compares last element with the preceding element. If the last element is less than that of preceding element swapping takes place. Then the preceding element is compared with that previous element. This process continues until the II and I elements are compared with each other. This is known as pass1.

This way the number of passes would be equal to size of array-1.

### प्रश्न 1. Sorting क्या होता है? ये कितने प्रकार के होते हैं?

उत्तर—Data structure में sorting वह प्रक्रिया है जिसके द्वारा हम Data को एक logical order में arrange करते हैं। यह logical order Ascending order भी हो सकता है और Descending order भी हो सकता है। यहाँ Ascending का अर्थ होता है बढ़ते क्रम में और descending का अर्थ होता है घटते क्रम में।

Sorting का अर्थ संबंध ढूँढ़ने (searching) से है। हमारी अपनी जिंदगी में बहुत सी चीजें होती हैं जिन्हें हम ढूँढ़ते हैं। जैसे—

❖ Google में कोई Topic.

❖ Book में कोई page.

❖ Exam में Roll no.



❖ Mobile में Contact No. , etc.

ये सभी चीजें जो होती हैं वह sorted (Arrange) होती हैं जिससे हम आसानी से उन्हें ढूँढ़ लेते हैं।

### Type of sorting

Sorting मुख्यतः दो प्रकार की होती है—

1. Internal sorting
2. External sorting

1. **Internal sorting** : इस sorting में sort किये जाने वाला सभी डेटा main memory में ही रहता है। ये निम्न प्रकार के होते हैं—

1. Bubble sort
2. Insertion sort
3. Quick sort
4. Heap sort
5. Selection sort

2. **External Sorting** : इस sorting में sort किये जाने वाला data secondary memory में रहता है क्योंकि data इतना ज्यादा होता है कि वह main memory में नहीं आ पाता है। External sorting एक ही प्रकार का होता है वह है Merge sort.

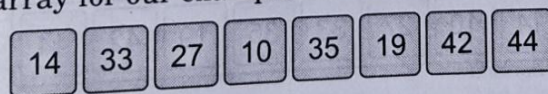
### Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

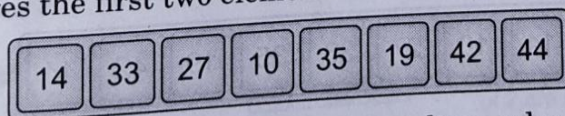
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$ , where  $n$  is the number of items.

### How Insertion Sort Works ?

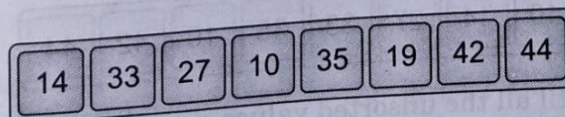
We take an unsorted array for our example.



Insertion sort compares the first two elements.

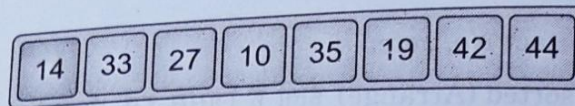


It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

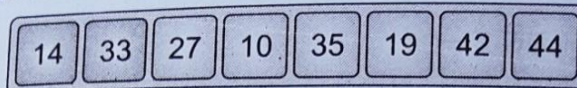


Insertion sort moves ahead and compares 33 with 27.

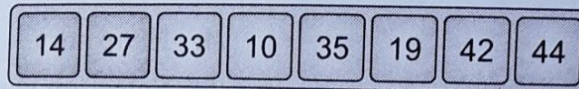




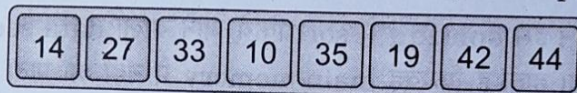
And finds that 33 is not in the correct position.



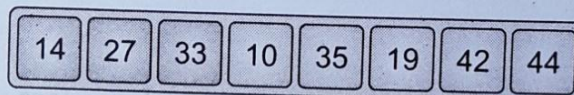
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



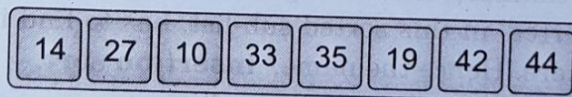
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



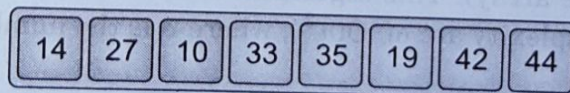
These values are not in a sorted order.



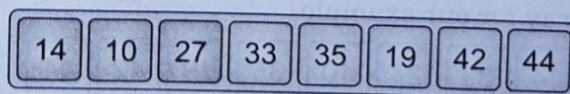
So we swap them.



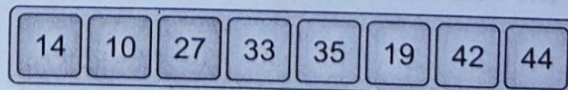
However, swapping makes 27 and 10 unsorted.



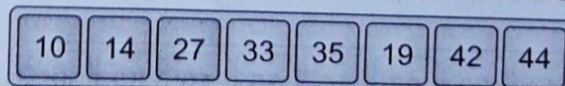
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.



### Implementation

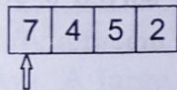
```

void insertion_sort (int A[ ], int n)
{
    for (int i = 0 ; i < n ; i++)
    {
        /*storing current element whose left side is checked for its correct position.*/
        int temp = A [i];
        int j = i;

        /* check whether the adjacent element in left side is greater or less than
        the current element.*/
        while (j>0 && temp < A[j-1])
        {
            //moving the left side element to one position forward.
            A[j] = A[j-1];
            j= j - 1;
        }
        //moving current element to its correct position.
        A [ j] = temp;
    }
}
    
```

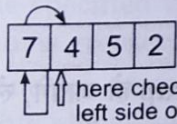
Take array A[]=[7,4,5,2].

Step 1 :

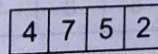


No element on left side of 7, so no change in its position.

Step 2 :

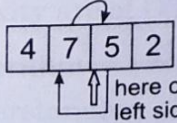


here checking on left side of 4.

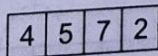


As  $7 > 4$ , therefore 7 will be moved forward and 4 will be moved to 7's position.

Step 3 :

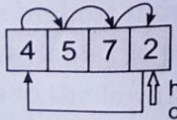


here checking on left side of 5.

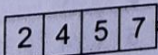


As  $7 > 5$ , 7 will be moved forward, but  $4 < 5$ , so no change in position of 4. And 5 will be moved to position of 7.

Step 4 :



here checking on left side of 2



As all the elements on left side of 2 are greater than 2, so all the elements will be moved forward and 2 will be shifted to position of 4

Since 7 is the first element, has no other element to be compared with, it remains at its position. Now when on moving towards 4, 7 is the largest element in the sorted list and greater than 4. So, move 4 to its correct position i.e. before 7. Similarly with 5, as 7 (largest element in the sorted list) is greater than 5, we will move 5 to its correct position. Finally for



2, all the elements on the left side of 2 (sorted list) are moved one position forward as all are greater than 2 and then 2 is placed in the first position. Finally, the given array will result in a sorted array.

**प्रश्न 2. Insertion sort क्या होता है? Example के द्वारा समझाइए तथा Algorithm भी लिखिए।**

**उत्तर—**Insertion sort एक simple sorting method है जिसमें पहले data का दूसरे data से compare किया जाता है। अगर पहला data दूसरे data से बड़ा है तो उन दोनों की value को आपस में बदल दिया जाता है फिर दूसरे value को तीसरे value से compare करते हैं। अगर तीसरा value दूसरे से छोटा है तो value बदलने के बाद पहले value से भी compare किया जाता है।

**Example :** माना कि हमारे पास निम्नलिखित Array है जिसे हमें short करना है।

**Pass 0 :**

|   |   |   |   |   |
|---|---|---|---|---|
| 5 | 3 | 8 | 4 | 6 |
| 0 | 1 | 2 | 3 | 4 |

**Pass 1 :** Insertion sort में पहले के दो elements को compare किया जाता है।

|         |   |   |   |   |
|---------|---|---|---|---|
| 5       | 3 | 8 | 4 | 6 |
| $5 > 3$ |   |   |   |   |

चूँकि 5, 3 से बड़ा तो ये आपस में अपना स्थान बदल (swap) लेगे।

**Pass 2 :**

|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 5 | 8 | 4 | 6 |
| 0 | 1 | 2 | 3 | 4 |

यहाँ पर हम 5 और 8 को compare करते हैं लेकिन यह पहले से ही sorted है इसलिए इनके स्थान में कोई परिवर्तन नहीं होगा।

**Pass 3 :**

|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 5 | 8 | 4 | 6 |
|---|---|---|---|---|

अब हम 8 और 4 को compare करेंगे। चूँकि  $8 > 4$  तो यह आपस में swap हो जायेंगे, लेकिन अब इसमें 5 और 4 unsorted हो जायेंगे इसलिए इन्हें भी आपस में swap करके sort कर लिया जाता है।

|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 5 | 8 | 4 | 6 |
| 0 | 1 | 2 | 3 | 4 |

|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 5 | 4 | 8 | 6 |
| 0 | 1 | 2 | 3 | 4 |

|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 4 | 5 | 8 | 6 |
| 0 | 1 | 2 | 3 | 4 |

**Pass 4 :** अब हम 8 और 6 को compare करते हैं चूँकि  $8 > 6$  तो आपस में अपना स्थान बदलेंगे। (swap होंगे)।



|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 4 | 5 | 8 | 6 |
| 0 | 1 | 2 | 3 | 4 |

Pass 5 : अन्त में हमें एक Sorted Array प्राप्त होगा।

|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 8 |
| 0 | 1 | 2 | 3 | 4 |

### Algorithm of Insertion sort

Insertion sort (ALMAXSIZE), item)

Step 1 : set  $k = 1$

Step 2 : for  $k = 1$  to  $(n - 1)$

Set  $temp = a[k]$

Set  $J = k - 1$

while  $temp < a[j]$  and  $(j \geq 0)$

set  $a[j+1] = a[j]$

[End of loop]

Assign the value of  $temp$  to  $a[j+1]$ .

Step 3 : Exit.

### Quick Short

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of  $O(n^2)$ , where  $n$  is the number of items.

### Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 : Choose the highest index value has pivot

Step 2 : Take two variables to point left and right of the list excluding pivot

Step 3 : Left points to the low index

Step 4 : Right points to the high

Step 5 : While value at left is less than pivot move right

Step 6 : While value at right is greater than pivot move left

Step 7 : If both step 5 and step 6 do not match swap left and right

Step 8 : If  $left \geq right$ , the point where they meet is new pivot



### Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quick sort as follows :

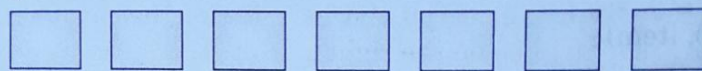
**Step 1 :** Make the right-most index value pivot

**Step 2 :** Partition the array using pivot value

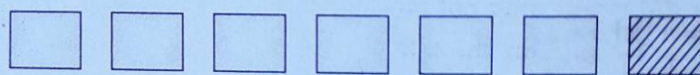
**Step 3 :** Quick sort left partition recursively

**Step 4 :** Quick sort right partition recursively

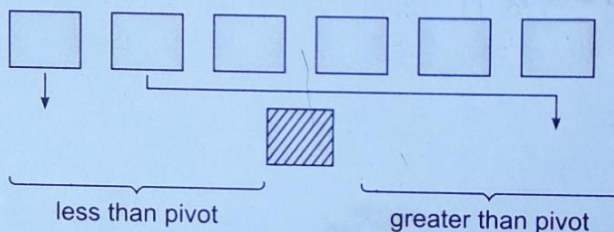
**Example :**



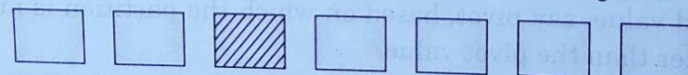
1. Start with an unsorted collection



2. Choose an element as the pivot



2. Using the pivot, partition the collection into 2 parts : elements smaller than the partition will be moved to the left/front of the partition, and elements larger than the pivot will be moved to the right/back.



3. continue choosing a pivot and partitioning (solving this problem recursively) until all the elements are sorted compared to the original pivot.

We'll choose the last element as the pivot for now. As it turns out, there are many different ways to choose a pivot element, and *what* you choose does matter—but more on that in a bit. It's pretty common to see implementations of quick sort with the last element as the pivot, so that's what we'll do here, too.

Okay, so we choose the last element as our pivot. Now, our quick sort algorithm will take all of the remaining, other elements, and reorder them so that all of the items *smaller* than our pivot and in front, or to the left of it, and all of the items *larger* than our pivot are behind, or to the right of it.

Since we know that quick sort is a divide and conquer algorithm, we also know that it's going to implement the *exact same logic* that we just saw on the two partitioned sublists! In other words, it's going to employ recursion here : it will choose a pivot element for each of the two sublists, and then divide each sublist into its own sublist, with two halves : a half that contains



all the elements smaller than the pivot, and another half that contains all the elements larger than the pivot. It'll continue calling itself *upon* itself recursively until it has only one element in each list—remember that one element in a list is, by definition, considered sorted.

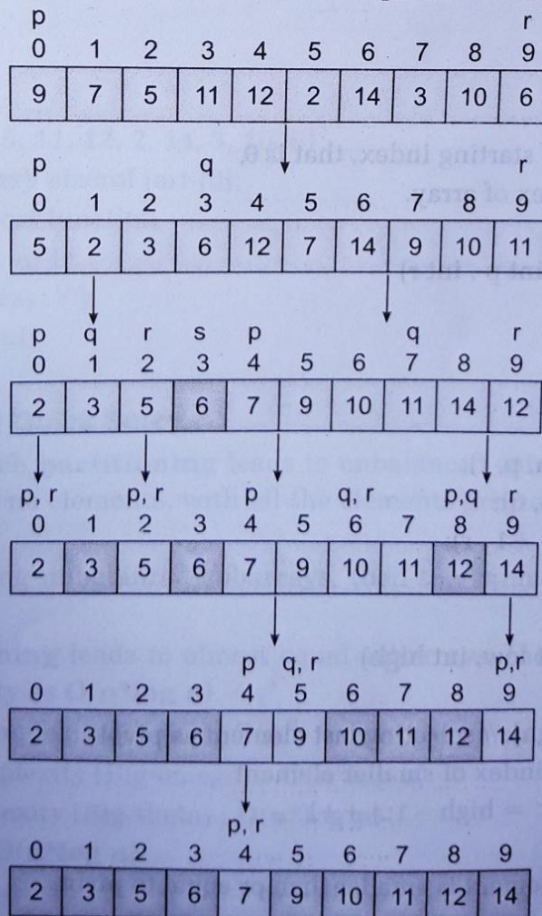
## How Quick Sorting Works ?

Following are the steps involved in quick sort algorithm :

1. After selecting an element as **pivot**, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the **pivot** will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the pivot element will be at its final **sorted** position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of **pivot** and elements on the right of **pivot**, and we perform **partitioning** on them by choosing a **pivot** in the subarrays.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.





In step 1, we select the last element as the pivot, which is 6 in this case, and call for partitioning, hence re-arranging the array in such a way that 6 will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a pivot for them, in the above diagram, we choose 3 as pivot for the left subarray and 11 as pivot for the right subarray.

And we again call for partitioning.

### Implementing Quick Sort Algorithm

Below we have a simple C program implementing the Quick sort algorithm :

```
// simple C program for Quick Sort
```

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

```
// to swap two numbers
```

```
void swap (int*a, int*b)
```

```
{
```

```
    int t=*a;
```

```
    *a=*b;
```

```
    *b = t;
```

```
}
```

```
/*
```

a [] is the array, p is starting index, that is 0,

and r is the last index of array.

```
*/
```

```
void quicksort (int a [ ], int p , int r)
```

```
{
```

```
    if (p< r)
```

```
    {
```

```
        int q;
```

```
        q = partition (a, p, r);
```

```
        quicksort (a, p, q);
```

```
        quicksort (a, q +1 , r);
```

```
    }
```

```
}
```

```
int partition (int a[ ], int low, int high)
```

```
{
```

```
    int pivot = arr [high]; // selecting last element as pivot
```

```
    int i = (low -1); // index of smaller element
```

```
    for (int j = low; j <= high - 1; j ++)
```

```
    {
```

```
        // If current element is smaller than or equal to pivot
```



```

    if (arr [j] <= pivot)
    {
        i ++;    // increment index of smaller element
        swap (& arr [i], &arr [j]);
    }
}
swap (&arr [i+1], & arr [high]);
return (i+1);
}
//function to print the array
void print_Array (int a [], int size)
{
    int i;
    for (i= 0; i < size ; i ++ )
    {
        printf ("%d", a [i]);
    }
    printf ("\n");
}

void main ( )
{
    int arr [ ]={9, 7, 5, 11, 12, 2, 14, 3, 10, 6};
    int n = size of (arr)/ sizeof (arr [0]);
    //call quicksort function
    quick sort (arr, 0, n-1);
    printf("Sorted array: \n");
    print_array (arr, n);
}

```

### Complexity Analysis of Quick Sort

For an array, in which **partitioning** leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the **pivot**, hence on the right side.

And if keep on getting unbalanced subarrays, then the running time is the worst case, which is  $O(n^2)$

Whereas if **partitioning** leads to almost equal subarrays, then the running time is the best, with time complexity as  $O(n \cdot \log n)$ .

Worst Case Time Complexity [Big-O] :  $O(n^2)$

Best Case Time Complexity [Big-omega] :  $O(n \cdot \log n)$

Average Time Complexity [Big-theta] :  $O(n \cdot \log n)$

Space Complexity :  $O(n \cdot \log n)$



**प्रश्न 3. Quick sort क्या होता है? इसको example के द्वारा समझाइये।**

**उत्तर—**Quick sort एक divide & conquer Algorithm पर आधारित sorting technique है। इस sorting technique में Array के elements को दो छोटे arrays में बाँटा जाता है।

इस Sorting में सबसे पहले list में से किसी भी elements को select किया जाता है जिसे हम pivot कहते हैं, pivot से छोटे elements इसके बाएँ (left side) तरफ रहेंगे जबकि pivot से बड़ा elements इसके दायीं (right side) तरफ रहेंगे।

Quick sort की औसत Complexity  $O(n \log n)$  है तथा इसकी worst case complexity  $O(n^2)$  है जहाँ  $n$ , elements की संख्या है। क्योंकि worst case में भी quick sort की complexity कम होती है इसलिए यह बहुत fast तथा efficient है।

### Algorithm for Quick sort

**Step 1 :** Array list में एक element को select करते हैं जिसे हम pivot value कहते हैं।

**Step 2 :** Element को इस प्रकार दुबारा Arrange करते हैं कि वे सभी elements जो pivot value से छोटी हैं वे Array के बायीं (left) तरफ रहती हैं और वे सभी elements जो pivot value से बड़ी होती हैं उन्हें array के दायीं (right-side) तरफ रखा जाता है और वे elements जो pivot के समान होते हैं उन्हें Array में किसी भी तरफ रखा जा सकता है।

**Step 3 :** Array के दोनों भागों को sort किया जाता है, दोनों भागों को दुबारा Quick sort Algorithm का प्रयोग करके short किया जाता है।

**Example : Complet of Quick sort :** Quick sort को समझने के लिए एक unsorted Array लेते हैं।

|   |    |   |    |   |
|---|----|---|----|---|
| 8 | 33 | 6 | 21 | 4 |
| 0 | 1  | 2 | 3  | 4 |

इस Array को sort करने के लिए (Ascending order में) निम्न steps को perform करते हैं—

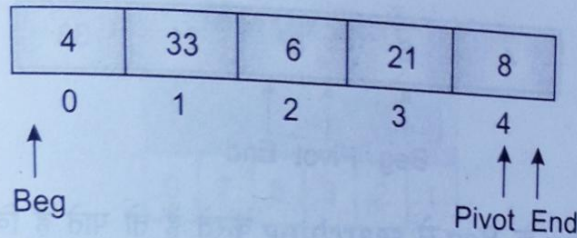
#### Pass First :

**Step 1 :** सबसे पहले Array के Index 0 (शून्य) को pivot (key) select करते हैं। Array के first Index को Beg और  $(n-1)$  Index को End से represent करेंगे।

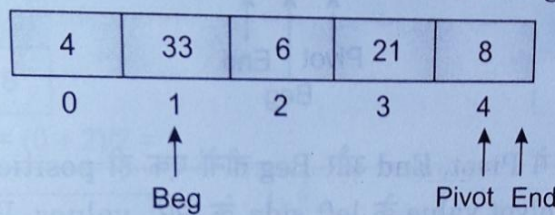
|       |     |   |    |     |
|-------|-----|---|----|-----|
| 8     | 33  | 6 | 21 | 4   |
| 0     | 1   | 2 | 3  | 4   |
| ↑     | ↑   |   |    | ↑   |
| Pivot | Beg |   |    | End |

**Step 2 :** अब हम Array के last index से यह search करते हैं कि pivot से कोई value छोटा तो नहीं है। जैसे pivot value 8, Array के last index की value 4 से बड़ा है इसलिए इनकी value swap (exchange) बदल दी जाएगी।

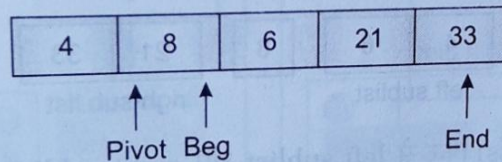




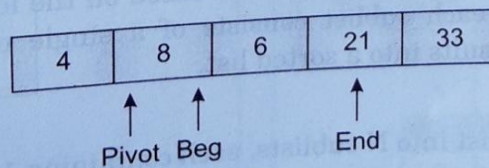
**Step 3 :** अब हम Array के staring (Beg) से search करना शुरू करेंगे कि pivot की value से कोई value बड़ी तो नहीं; जैसे कि यहाँ पर pivot की value 8, Array की Beg की value 4 से बड़ी है इसलिए Beg की value में एक (Increment) बढ़ा दिया जायेगा और किसी प्रकार का कोई change (बदलाव) नहीं किया जायेगा।



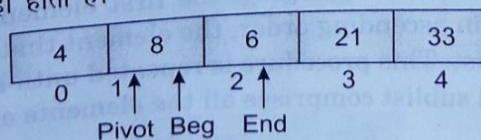
**Step 4 :** अब pivot की value 8, Array की starting value 33 से छोटा है तब इनकी value को आपस में बदल देंगे।



**Step 5 :** फिर हम Array के end से searching शुरू करते हैं तो पाते हैं कि pivot की value 8, Array के last index से छोटा है तब end की value में एक (decrement) घटा देंगे। और list में किसी प्रकार का कोई changes नहीं होगा।

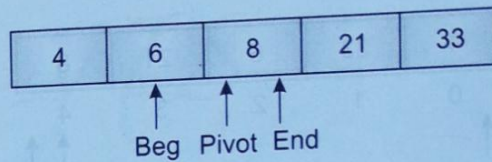


**Step 6 :** फिर Array के end से searching शुरू करते हैं तो पाते हैं कि pivot की value 8, Array की end की value 21 से छोटा है, इसलिए end की value में एक (1) का decrement (घटा) करते हैं और list में किसी प्रकार का कोई changes नहीं होता है।

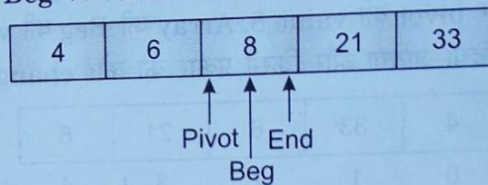


**Step 7 :** अब फिर Array के end से searching start करते हैं जिसमें पाते हैं कि pivot की value 8, Array के end की value 6 से बड़ी है इसलिए दोनों की values को आपस में बदल देते हैं।



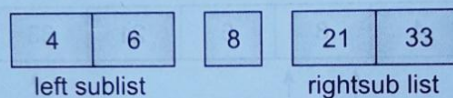


**Step 8 :** अब list को Array के Beg से searching करते हैं तो पाते हैं कि pivot की value 8, Array के Beg की value से बड़ी है इसलिए Beg की value को 1. Increment करेंगे और list में कोई changes नहीं होगा।



ऐसे situation में, जब list में Pivot, End और Beg तीनों एक ही position पर आ जाते हैं तब first pass complete हो जाएगा। इस प्रकार pivot value के left side के सारे values, Pivot से छोटे होंगे और Pivot के right side के सारे values, Pivot से बड़े होंगे।

इस प्रकार list, left sublist और right sublist में बंट जाता है।



इसी प्रकार पूरा तरीका (process) फिर से left sublist और right sublist पर perform होगा।

## Merge Sort

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sub-lists in a manner that results into a sorted list.

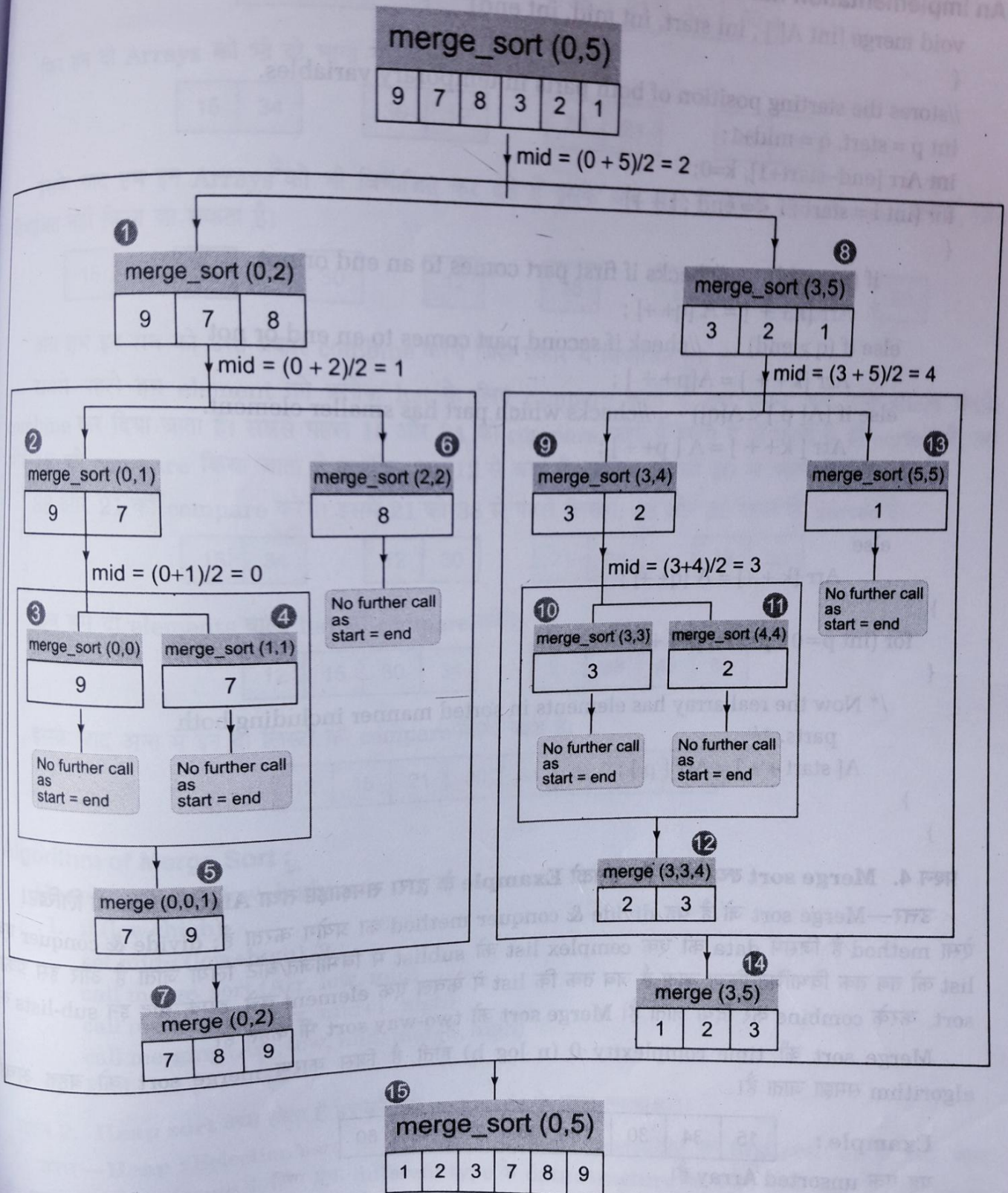
### Idea :

- ❖ Divide the unsorted list into  $N$  sublists, each containing 1 element.
- ❖ Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements.  $N$  will now convert into  $N/2$  lists of size 2.
- ❖ Repeat the process till a single sorted list is obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.



# Merge Sort





An implementation has been provided below :

```
void merge (int A[ ], int start, int mid, int end)
{
    //stores the starting position of both parts in temporary variables.
    int p = start, q = mid+1;
    int Arr [end-start+1], k=0;
    for (int i = start ; i <= end ; i++)
    {
        if (p > mid) //checks if first part comes to an end or not.
            Arr [k++] = A [q++];
        else if (q > end) //check if second part comes to an end or not
            Arr [k++] = A [p++];
        else if (A [p] < A [q]) //checks which part has smaller element.
            Arr [k++] = A [p++];
        else
            Arr [k++] = A [q++];
    }
    for (int p=0 ; p < k ; p++)
    {
        /* Now the real array has elements in sorted manner including both
        parts. */
        A [start++] = Arr [p];
    }
}
```

**प्रश्न 4. Merge sort क्या होता है? इसको Example के द्वारा समझाइए तथा Algorithm भी लिखिए।**

उत्तर—Merge sort जो है वह divide & conquer method का प्रयोग करता है; divide & conquer एक ऐसा method है जिसमें data की एक complex list को sublist में विभाजित/बाँट लिया जाता है और इस प्रकार list को तब तक विभाजित किया जाता है जब तक कि list में केवल एक element बचे, इसके बाद इन sub-lists को sort. करके combine कर दिया जाता है। Merge sort को two-way sort भी कहते हैं।

Merge sort की time complexity  $O(n \log n)$  होती है जिस कारण merge sort को बहुत अच्छी algorithm समझा जाता है।

**Example :**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 15 | 34 | 30 | 12 | 38 | 21 | 43 | 50 |
|----|----|----|----|----|----|----|----|

यह एक unsorted Array है।

जैसा कि हम जानते हैं कि merge sort में सबसे पहले पूरे Array को आधे भाग में विभाजित किया जाता है, हमारे पास इस Array में 8 elements हैं तथा इस Array को दो भागों में विभाजित किया जाता है जिनमें कि 4-4 elements होंगे।



|    |    |    |    |
|----|----|----|----|
| 15 | 34 | 30 | 12 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| 38 | 21 | 43 | 50 |
|----|----|----|----|

फिर इन दो Arrays को भी दो भागों में विभाजित किया जाता है।

|    |    |
|----|----|
| 15 | 34 |
|----|----|

|    |    |
|----|----|
| 30 | 12 |
|----|----|

|    |    |
|----|----|
| 38 | 21 |
|----|----|

|    |    |
|----|----|
| 43 | 50 |
|----|----|

इसके बाद हम इन Arrays को भी विभाजित कर देते हैं इसके बाद list में केवल एक element बचेगा, जिसे विभाजित नहीं किया जा सकता है।

|    |
|----|
| 15 |
|----|

|    |
|----|
| 34 |
|----|

|    |
|----|
| 30 |
|----|

|    |
|----|
| 12 |
|----|

|    |
|----|
| 38 |
|----|

|    |
|----|
| 21 |
|----|

|    |
|----|
| 43 |
|----|

|    |
|----|
| 50 |
|----|

अब हम इन सब को उसी प्रकार combine करेंगे जिस प्रकार ये विभाजित हुए थे।

सबसे पहले हम element को प्रत्येक list के लिए compare करते हैं तथा उसके बाद इन्हें short करके combine कर दिया जाता है। सबसे पहले 15 और 34 को compare करते हैं परन्तु ये तो पहले से ही sorted हैं, 30 और 12 को compare किया जाता है क्योंकि 30, 12 से बड़ा है इसलिए 12 को 30 से पहले लिखेंगे।

38 और 21 को compare करेंगे। इसमें 21 को 38 से पहले लिखेंगे; 43 और 50 पहले से sorted हैं।

|    |    |
|----|----|
| 15 | 34 |
|----|----|

|    |    |
|----|----|
| 12 | 30 |
|----|----|

|    |    |
|----|----|
| 21 | 38 |
|----|----|

|    |    |
|----|----|
| 43 | 50 |
|----|----|

अब हम दो elements वाली list को compare करेंगे।

|    |    |    |    |
|----|----|----|----|
| 12 | 15 | 30 | 34 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| 21 | 38 | 43 | 50 |
|----|----|----|----|

इसके बाद अन्त में इन दो लिस्टों को compare किया जाता है।

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 12 | 15 | 21 | 30 | 34 | 38 | 43 | 50 |
|----|----|----|----|----|----|----|----|

### Algorithm of Merge Sort :

Merge\_sort (Arr, low, high)

1. if (low < high)  
 set mid = (low + high) / 2  
 call merge\_sort (Arr, low, mid)  
 call merge\_sort (Arr, mid + 1, high)  
 call merging (Arr, low, mid, mid + 1, high)
2. End.

**प्रश्न 2. Heap sort क्या होता है इसको एक example के द्वारा समझाइये।**

**उत्तर—Heap :** Selection sort की तरह सबसे बड़ा या छोटा element को first position पर रखा जाता है। इस प्रकार के ordering के लिए एक different type के data structure का प्रयोग किया जाता है जिसे Heap कहाँ जाता है जो कि एक complete binary tree के रूप (form) में दिखाई देता है।

### Types of Heap :

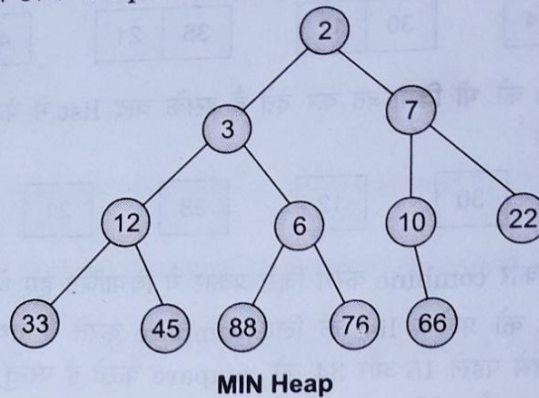
Heap दो प्रकार के होते हैं—



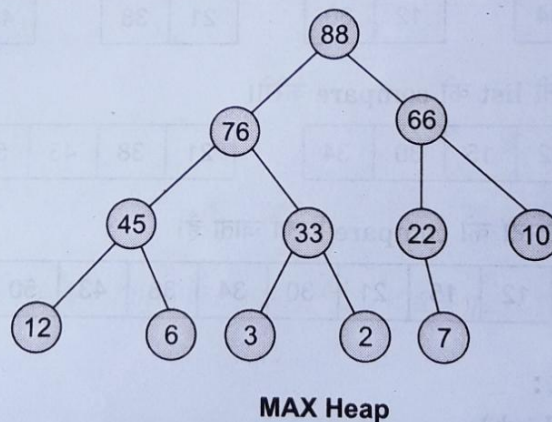
## 2. Max Heap

## 1. Min Heap

1. **Min Heap** : Min Heap या Ascending Heap. इस प्रकार का heap है जिसमें प्रत्येक Parent node, अपने child node से छोटा या equal (समान) होगा। जैसे कि



2. **Max Heap** : Max heap या descending heap, इस प्रकार का heap है जिसमें parent node की value, इसके child node की value से बड़ी या equal (समान) होती है।



**Example :** Heap sort को समझने के लिए एक unsorted array को लेते हैं।

|   |    |   |    |    |    |    |    |    |    |
|---|----|---|----|----|----|----|----|----|----|
| 9 | 11 | 6 | 45 | 22 | 10 | 12 | 90 | 67 | 17 |
| 0 | 1  | 2 |    | 4  | 5  | 6  | 7  | 8  | 9  |

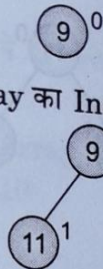
Array में elements को एक-एक करके tree में represent किया जाएगा। यदि Array का कोई element, अपने parent node से बड़ा है तो वह जिस location पर store होगा इसका पता  $(i-1)/2$  से लगेगा। तब इसकी value को parent की value से swap कर देंगे। दूसरे element को अपने new parent node से compare किया जाता है और उनकी value swap (आपस में बदली जाती है) की जाती है। यदि value, parent node से बड़ी है, तो इस Process को तब तक किया जाता है जब तक कि कोई और swapping की जरूरत न हो।

दिया गया Array sort करते हैं।

**Step 1 :** पहला element-9 Array के Index [0] पर store है। इसलिए इसको Tree में 0th position पर represent करेंगे।



Step 2 : अब दूसरा element tree में (Array का Index [1]) Root 9 का left child बनेगा।

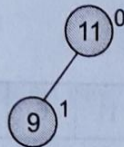


अब 11 को, 9 से compare किया जाता है, जो कि  $(i-1)/2, (1-1)/2 = 0$ .

$$\text{Array}[0] < \text{Array}[1]$$

$$9 < 11$$

अब इनकी value को आपस में swap कर (बदल) दी जायेगी, तो Tree इस प्रकार दिखाई देगा—



और Array

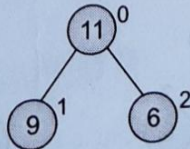
|    |   |   |    |    |    |    |    |    |    |
|----|---|---|----|----|----|----|----|----|----|
| 11 | 9 | 6 | 45 | 22 | 10 | 12 | 90 | 67 | 17 |
| 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  |

Step 3 : अब Array का तीसरा element 6 अपने parent node से compare होगा जो कि  $(i-1)/2 = (2-1)/2 = 1/2 = 0$  पर है।

$$\text{Array}[0] > \text{Array}[2]$$

$$11 > 6$$

तब ये swap नहीं होंगे और Array में कोई changes नहीं होगा, तो tree और Array इस प्रकार दिखाई देंगे—



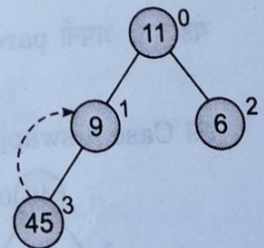
|    |   |   |    |    |    |    |    |    |    |
|----|---|---|----|----|----|----|----|----|----|
| 11 | 9 | 6 | 45 | 22 | 10 | 12 | 90 | 67 | 17 |
| 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  |

Step 4 : Array का 4th element 45 है जो कि अपने parent node से compare होगा जो  $(i-1)/2 = (3-1)/2 = 2/2 = 1$  पर है।

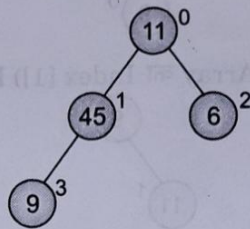
$$\text{Array}[1] < \text{Array}[3]$$

$$9 < 45$$

तब इनकी value swap करेंगे।







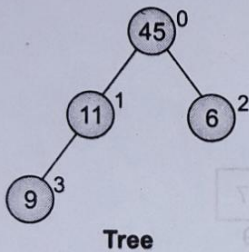
यह पुनः अपने parent node से compare होगा जो कि  $(i-1)/2 = (1-1)/2 = 0$  है।

Array [0] > Array [1]

11 > 45

तब इनकी value पुनः swap होगी।

तो Tree और Array इस प्रकार होंगे।



|    |    |   |   |    |    |    |    |    |    |
|----|----|---|---|----|----|----|----|----|----|
| 45 | 11 | 6 | 9 | 22 | 10 | 12 | 90 | 67 | 17 |
| 0  | 1  | 2 | 3 | 4  | 5  | 6  | 7  | 8  | 9  |

Array

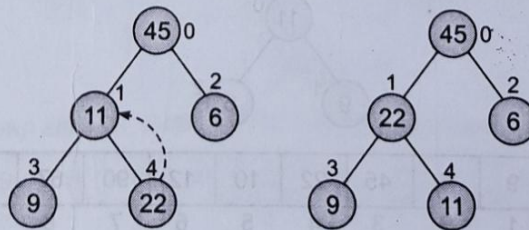
**Step 5 :** अगला element 22 है जो कि अपने parent node से compare होगा जो कि

$(i-1)/2 = (4-1)/2 = 3/2 = 1$

Array [1] < Array [4]

11 < 22

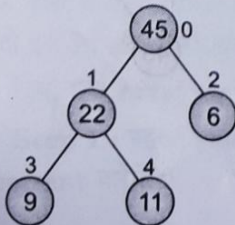
तब इनकी value swap होगी।



यह पुनः अपनी parent node से compare होगा जो कि  $(i-1)/2 = (1-1)/2 = 0$

Array [0] > Array [1]

इस Case में swapping नहीं होगी। Tree और Array इस प्रकार होंगे—



|    |    |   |   |    |    |    |    |    |    |
|----|----|---|---|----|----|----|----|----|----|
| 45 | 22 | 6 | 9 | 11 | 10 | 12 | 90 | 67 | 17 |
| 0  | 1  | 2 | 3 | 4  | 5  | 6  | 7  | 8  | 9  |

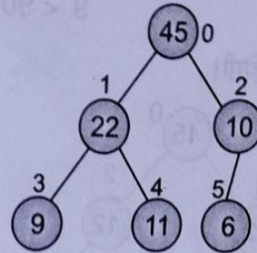
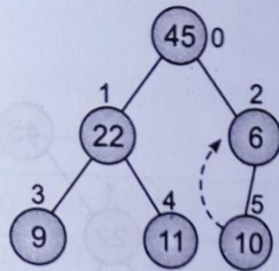


**Step 6 :** अब element 10 का अपने Parent node से compare किया जाएगा जो कि  $(i-1)/2 = (5-1)/2 = 4/2 = 2$  पर है।

$$\text{Array}[2] < \text{Array}[5]$$

$$6 < 10$$

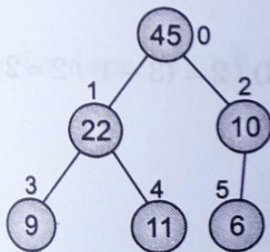
तब इनकी value आपस में swap होगी।



यह पुनः अपने parent node से compare होगा जो कि  $(i-1)/2 = (2-1)/2 = 1/2 = 0$

$$\text{Array}[0] > \text{Array}[1]$$

इस case में swapping नहीं होगा। तो Tree और Array इस प्रकार होंगे—



|    |    |    |   |    |   |    |    |    |    |
|----|----|----|---|----|---|----|----|----|----|
| 45 | 22 | 10 | 9 | 11 | 6 | 12 | 90 | 67 | 17 |
| 0  | 1  | 2  | 3 | 4  | 5 | 6  | 7  | 8  | 9  |

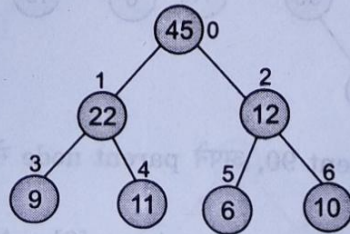
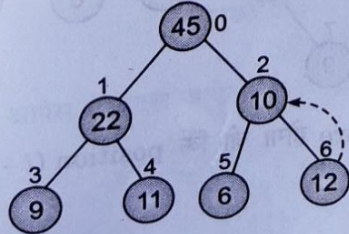
**Step 7 :** अब अगला element 12 है जो कि अपने parent node से compare होगा जो कि

$$(i-1)/2 = (6-1)/2 = 5/2 = 2 \text{ पर है।}$$

$$\text{Array}[2] < \text{Array}[6]$$

$$10 < 12$$

इस case में इनकी value swap होगी।



पुनः इसका अपने Parent node से compare होगा जो कि  $(i-1)/2 = (2-1)/2 = 1/2 = 0$

$$\text{Array}[0] > \text{Array}[2]$$

$$45 > 10$$

इस case में कोई changes नहीं होगा तो Tree और Array इस प्रकार होंगे—



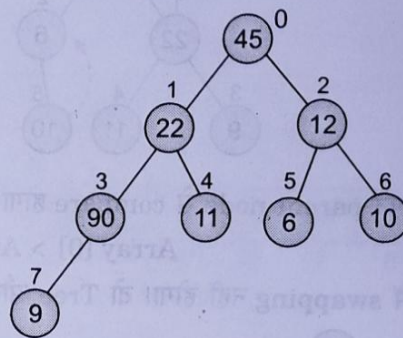
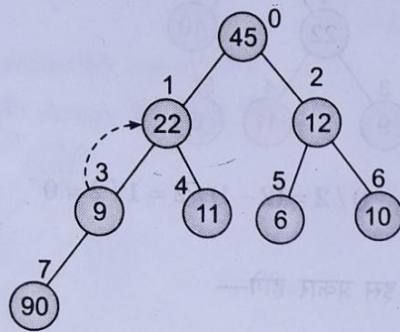
|    |    |    |   |    |   |    |    |    |    |
|----|----|----|---|----|---|----|----|----|----|
| 45 | 22 | 12 | 9 | 11 | 6 | 10 | 90 | 67 | 17 |
| 0  | 1  | 2  | 3 | 4  | 5 | 6  | 7  | 8  | 9  |

**Steps 8 :** अगला Element 90 है जो कि अपने Parent node से compare होगा जो कि position  $(i-1)/2 = (7-1)/2 = 6/2 = 3$  पर है।

Array [3] < Array [7]

9 < 90

तब इनकी value swap होगी।

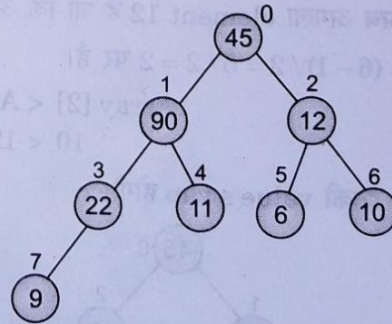
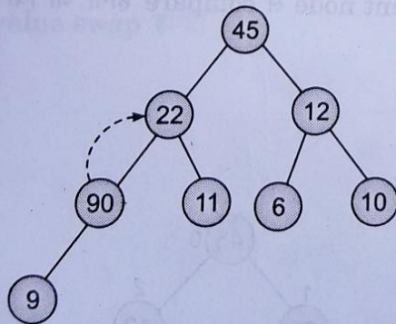


पुनः यह value अपने parent node से compare होगा जो कि position  $(i-1)/2 = (3-1)/2 = 2/2 = 1$  पर है।

Array [1] < Array [3]

22 < 90

पुनः ये values आपस में swap होगी।

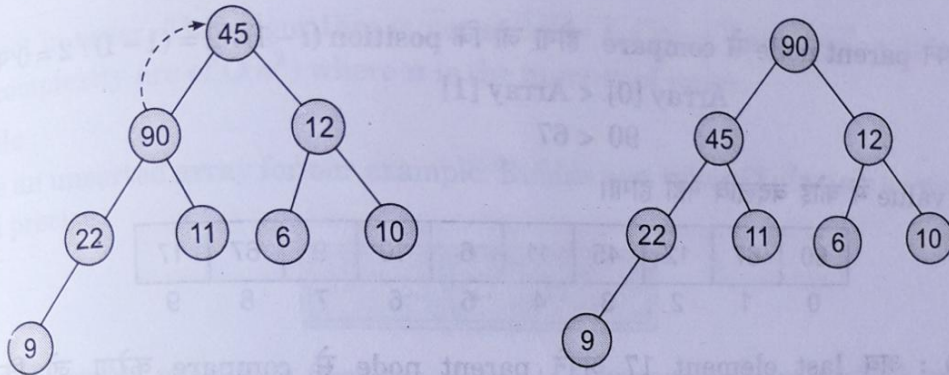


पुनः यह element 90, अपने parent node से compare होगा जो कि position  $(i-1)/2 = (1-1)/2 = 0/2 = 0$  पर है।

Array [0] < Array [1]

तब पुनः इनकी values आपस में swap होगी।





अब Array इस प्रकार होगा—

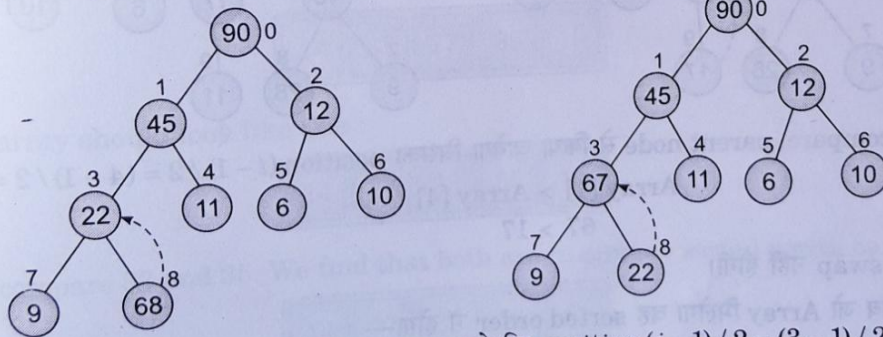
|    |    |    |    |    |   |    |   |    |    |
|----|----|----|----|----|---|----|---|----|----|
| 90 | 45 | 12 | 22 | 11 | 6 | 10 | 9 | 67 | 17 |
| 0  | 1  | 2  | 3  | 4  | 5 | 6  | 7 | 8  | 9  |

**Step 9 :** अगला element 67 है जिसका अपने parent node से compare होगा जो कि position  $(i-1)/2 = (8-1)/2 = 7/2 = 3$  पर है।

$$\text{Array}[3] < \text{Array}[8]$$

$$22 < 67$$

तब ये values आपस में बदल जायेंगी।

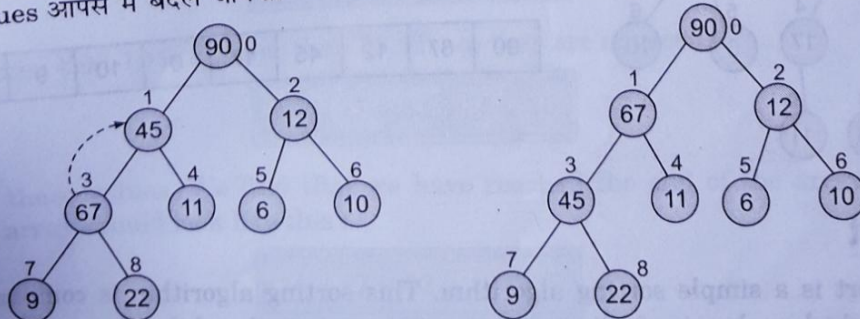


पुनः अपने parent node से compare किया जाएगा जो कि position  $(i-1)/2 = (3-1)/2 = 2/2 = 1$

$$\text{Array}[1] < \text{Array}[3]$$

$$45 < 67$$

पुनः ये values आपस में बदल जायेंगी।





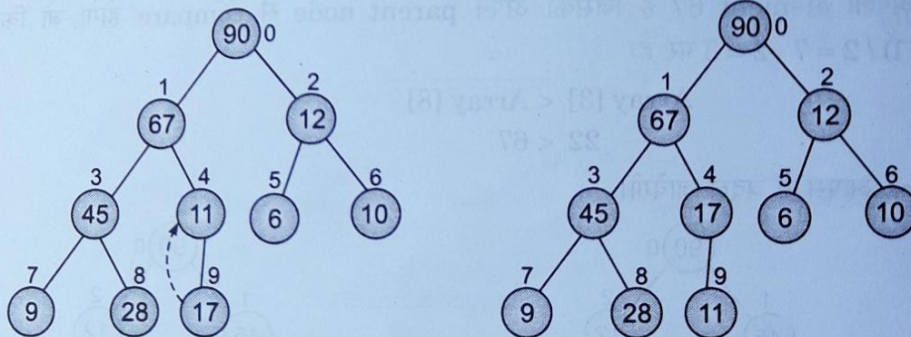
67 पुनः अपने parent node से compare होगा जो कि position  $(i-1)/2 = (1-1)/2 = 0$  पर है।  
 $\text{Array}[0] < \text{Array}[1]$   
 $90 < 67$

अब इनकी value में कोई बदलाव नहीं होगा।

|    |    |    |    |    |   |    |   |    |    |
|----|----|----|----|----|---|----|---|----|----|
| 90 | 67 | 12 | 45 | 11 | 6 | 10 | 9 | 67 | 17 |
| 0  | 1  | 2  | 3  | 4  | 5 | 6  | 7 | 8  | 9  |

**Step 10 :** अब last element 17 अपने parent node से compare करेगा जो कि position  $(i-1)/2 = (9-1)/2 = 8/2 = 4$  पर है।  
 $\text{Array}[4] < \text{Array}[9]$   
 $11 < 17$

तब इनकी value swap होगी।

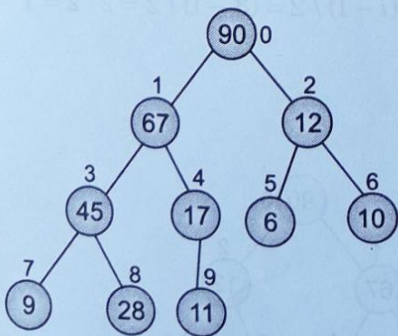


पुनः इसका compare, parent node से किया जायेगा जिसका position  $(i-1)/2 = (4-1)/2 = (3/2) = 1$

$\text{Array}[1] > \text{Array}[4]$   
 $67 > 17$

अब value swap नहीं होगी।

इस प्रकार अब जो Array मिलेगा वह sorted order में होगा—



|    |    |    |    |    |   |    |   |    |    |
|----|----|----|----|----|---|----|---|----|----|
| 90 | 67 | 12 | 45 | 17 | 6 | 10 | 9 | 28 | 11 |
|----|----|----|----|----|---|----|---|----|----|

## Bubble Sort

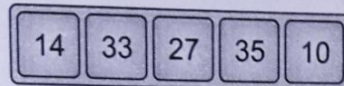
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped



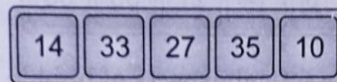
if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where  $n$  is the number of items.

### Example :

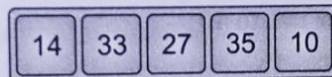
We take an unsorted array for our example. Bubble sort takes  $O(n^2)$  time so we're keeping it short and precise.



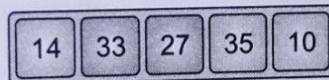
Bubble sort starts with very first two elements, comparing them to check which one is greater.



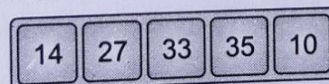
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



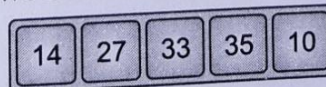
We find that 27 is smaller than 33 and these two values must be swapped.



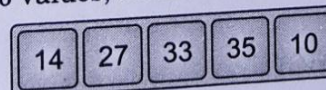
The new array should look like this :



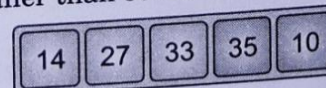
Next we compare 33 and 35. We find that both are in already sorted positions.



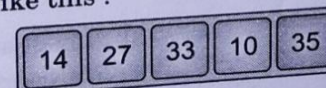
Then we move to the next two values, 35 and 10.



We know then that 10 is smaller than 35. Hence they are not sorted.

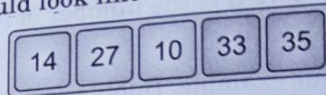


We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this :

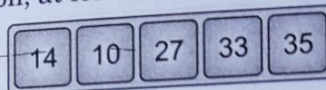




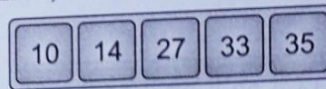
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this :



Notice that after each iteration, at least one value moves at the end.



And when there is swap required, bubble sort learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

### Algorithm of Bubble Sort

Step 1 : Repeat Steps 2 and 3 for  $i=1$  to 10

Step 2 : Set  $j=1$

Step 3 : Repeat while  $j \leq n$

(A) if  $a[i] < a[j]$

Then interchange  $a[i]$  and  $a[j]$

[End of if]

(B) Set  $j = j+1$

[End of inner Loop]

[End of Step 1 Outer Loop]

Step 4 : Exit

### Implementation in C

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

```
#include <stdbool.h>
```

```
#define MAX 10
```

```
int list[MAX] = {1,8,4,6,0,3,5,2,7,9};
```

```
void display()
```

```
{
```

```
    int i;
```

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

```
    // navigate through all items
```

```
    for (i = 0; i < MAX; i++)
```

```
{
```

```
        printf("%d",list[i]);
```



```
}

printf("\n");
}

void bubbleSort()
{
    int temp;
    int i,j;

    bool swapped = false;
    // loop through all numbers
    for (i = 0; i < MAX-1; i++)
    {
        swapped = false;

        // loop through numbers falling ahead
        for(j = 0; j < MAX-1-i; j++)
        {
            printf(" Items compared; [%d, %d ]", list[j], list[j+1]);

            // check if next number is lesser than current no
            // swap the numbers.
            // (Bubble up the highest number)

            if(list[j] > list[j+1])
            {
                temp = list[j];
                list[j] = list[j+1];
                list[j+1] = temp;
                swapped = true;
                printf(" => swapped [%d, %d]\n",list[j],list[j+1]);
            }
            else
            {
                printf("=> not swapped\n");
            }
        }

        // if no number was swapped that means
        // array is sorted now, break the loop.
        if(!swapped)
```



```

    {
        break;
    }

    printf("Iteration %d#: ",(i+1));
    display();
}
}

void main()
{
    printf("Input Array: ");
    display();
    printf("\n");
    bubbleSort();
    printf("\nOutput Array:");
    display();
}

```

If we compile and run the above program, it will produce the following result :

### Output

Input Array : [ 1 8 4 6 0 3 5 2 7 9 ]

Items compared: [ 1, 8 ] => not swapped

Items compared: [ 8, 4 ] => swapped [4, 8]

Items compared: [ 8, 6 ] => swapped [6, 8]

Items compared: [ 8, 0 ] => swapped [0, 8]

Items compared: [ 8, 3 ] => swapped [3, 8]

Items compared: [ 8, 5 ] => swapped [5, 8]

Items compared: [ 8, 2 ] => swapped [2, 8]

Items compared: [ 8, 7 ] => swapped [7, 8]

Items compared: [ 8, 9 ] => not swapped

Iteration 1#: [ 1 4 6 0 3 5 2 7 8 9 ]

Items compared: [ 1, 4 ] => not swapped

Items compared: [ 4, 6 ] => not swapped

Items compared: [ 6, 0 ] => swapped [0, 6]

Items compared: [ 6, 3 ] => swapped [3, 6]

Items compared: [ 6, 5 ] => swapped [5, 6]

Items compared: [ 6, 2 ] => swapped [2, 6]

Items compared: [ 6, 7 ] => not swapped

Items compared: [ 7, 8 ] => not swapped



Iteration 2#: [1 4 0 3 5 2 6 7 8 9]

Items compared: [ 1, 4 ] => not swapped

Items compared: [ 4, 0 ] => swapped [0, 4]

Items compared: [ 4, 3 ] => swapped [3, 4]

Items compared: [ 4, 5 ] => not swapped

Items compared: [ 5, 2 ] => swapped [2, 5]

Items compared: [ 5, 6 ] => not swapped

Items compared: [ 6, 7 ] => not swapped

Iteration 3#: [1 0 3 4 2 5 6 7 8 9]

Items compared: [ 1, 0 ] => swapped [0, 1]

Items compared: [ 1, 3 ] => not swapped

Items compared: [ 3, 4 ] => not swapped

Items compared: [ 4, 2 ] => swapped [2, 4]

Items compared: [ 4, 5 ] => not swapped

Items compared: [ 5, 6 ] => not swapped

Iteration 4#: [0 1 3 2 4 5 6 7 8 9]

Items compared: [ 0, 1 ] => not swapped

Items compared: [ 1, 3 ] => not swapped

Items compared: [ 3, 2 ] => swapped [2, 3]

Items compared: [ 3, 4 ] => not swapped

Items compared: [ 4, 5 ] => not swapped

Iteration 5#: [0 1 2 3 4 5 6 7 8 9]

Items compared: [ 0, 1 ] => not swapped

Items compared: [ 1, 2 ] => not swapped

Items compared: [ 2, 3 ] => not swapped

Items compared: [ 3, 4 ] => not swapped

Output Array: [0 1 2 3 4 5 6 7 8 9]





## SEARCHING

Searching is a process of locating a particular element present in a given set of elements. The element may be a record, a table, or a file.

A search algorithm is an algorithm that accepts an argument 'a' and tries to find an element whose value is 'a'. It is possible that the search for a particular element in a set is unsuccessful if that element does not exist. There are a number of techniques available for searching.

Some of the standard searching techniques that are being followed in data structure are listed below :

1. Linear Search or Sequential Search
2. Binary Search

### Linear Search

This is the simplest method for searching. In this technique of searching, the element to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

As against this, searching in case of unsorted list also begins from the 0th element and continues until the element or the end of the list is reached.

|    |   |   |    |    |    |    |
|----|---|---|----|----|----|----|
| 10 | 1 | 9 | 11 | 46 | 20 | 16 |
|----|---|---|----|----|----|----|

One-Dimensional Array having 7 Elements

### Example :

The list given below is the list of elements in an unsorted array. The array contains 10 elements. Suppose the element to be searched is '46', so 46 is compared with all the elements starting from the 0<sup>th</sup> element and searching process ends where 46 is found or the list ends.

The performance of the linear search can be measured by counting the comparisons done to find out an element. The number of comparisons is  $O(n)$ .

Algorithm analysis should begin with a clear statement of the task to be performed. This allows us both to check that the algorithm is correct and to ensure that the algorithms we are comparing perform the same task.



Although there are many ways that algorithms can be compared, we will focus on two that are of primary importance to many data processing algorithms :

- ❖ *time complexity* : how the number of steps required depends on the size of the input.
- ❖ *space complexity* : how the amount of extra memory or storage required depends on the size of the input.

The linear search can be applied for both unsorted & sorted list.

1. Linear search for Unsorted list
2. Linear search for sorted list

### 1. Linear Search for Unsorted list :

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.

Linear search is implemented using following steps :

- ❖ **Step 1** : Read the search element from the user.
- ❖ **Step 2** : Compare, the search element with the first element in the list.
- ❖ **Step 3** : If both are matching, then display "Given element found!!!" and terminate the function.
- ❖ **Step 4** : If both are not matching, then compare search element with the next element in the list.
- ❖ **Step 5** : Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- ❖ **Step 6** : If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

### Example :

Consider the following list of elements and search element :

|                |    |    |    |    |    |    |    |    |
|----------------|----|----|----|----|----|----|----|----|
|                | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| list           | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |
| search element | 12 |    |    |    |    |    |    |    |

**Step 1** : search element (12) is compared with first element (65)

|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |
| 12   |    |    |    |    |    |    |    |    |

Both are not matching. So move to next element.

**Step 2** : search element (12) is compared with next element (20)



|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element.

**Step 3 :** search element (12) is compared with next element (10)

|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element.

**Step 4 :** search element (12) is compared with next element (55)

|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element.

**Step 5 :** search element (12) is compared with next element (32)

|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element.

**Step 6 :** search element (12) is compared with next element (12)

|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are matching. So we stop comparing and display element found at index 5.

## 2. Linear search for sorted list :

In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned. The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.

All the elements preceding the search element are traversed before the search element is traversed. *i.e.* if the element to be searched is in position 10, all elements from 1-9 are checked before 10.

**Algorithm :** Linear search implementation

```
bool linear_search (int*list, int size, int key, int*rec)
```

```
{
```

```
    //Basic Linear search
```

```
    bool found=false;
```



```

int i;
for (i = 0; i < size; i++)
{
    if (key == list[i])
        break;
}
if (i < size)
{
    found = true;
    rec = &list[i];
}
return found;
}

```

The code searches for the element through a loop starting from 0 to n. The loop can terminate in one of two ways. If the index variable *i* reaches the end of the list, the loop condition fails. If the current item in the list matches the key, the loop is terminated early with a break statement. Then the algorithm tests the index variable to see if it is less than that size (thus the loop was terminated early and the item was found), or not (and the item was not found).

#### Example :

Assume the element 45 is searched from a sequence of sorted elements 12, 18, 25, 36, 45, 48, 50. The Linear search starts from the first element 12, since the value to be searched is not 12 (value 45), the next element 18 is compared and is also not 45, by this way all the elements before 45 are compared and when the index is 5, the element 45 is compared with the search value and is equal, hence the element is found and the element position is 5.

| List |    |    |    |    |    |    | i | Result of comparison |
|------|----|----|----|----|----|----|---|----------------------|
| 12   | 18 | 25 | 36 | 45 | 48 | 50 | 1 | 12 <> 45 : false     |
| 12   | 18 | 25 | 36 | 45 | 48 | 50 | 2 | 18 <> 45 : false     |
| 12   | 18 | 25 | 36 | 45 | 48 | 50 | 3 | 25 <> 45 : false     |
| 12   | 18 | 25 | 36 | 45 | 48 | 50 | 4 | 36 <> 45 : false     |
| 12   | 18 | 25 | 36 | 45 | 48 | 50 | 5 | 45 = 45 : true       |

### Linear Search Program in C Programming Language

```

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

void main()
{
    int list[20], size, i, sElement;

```



```
printf("Enter size of the list:");
scanf("%d", &size);
```

```
printf("Enter any%d integer values :",size);
for (i=0;<size;i++)
(i = 0; i < size; i++)
scanf("%d",&list[i]);
```

```
printf("Enter the element to be Searched: ");
scanf("%d", sElement);
```

```
//Linear Search Logic
```

```
for (i=0;i<size;i++)
```

```
{
    if (sElement==list [i])
    {
        printf("Element is found at %d index",i);
        break;
    }
}
```

```
if(i==size)
```

```
printf("Given element is not found in the list!!!");
getch ();
}
```

## 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 elements. That means, binary search can be used only with list of elements which are already arranged in an order. The binary search can not be used for list of elements which are 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 sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist 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".

Binary search is implemented using following steps :

- ❖ **Step 1 :** Read the search element from the user.
- ❖ **Step 2 :** Find the middle element in the sorted list.
- ❖ **Step 3 :** Compare, the search element with the middle element in the sorted list.



- ❖ **Step 4 :** If both are matching, then display "Given element found!!" and terminate the function.
- ❖ **Step 5 :** If both are not matching, then check whether the search element is smaller or larger than middle element.
- ❖ **Step 6 :** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- ❖ **Step 7 :** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- ❖ **Step 8 :** Repeat the same process until we find the search element in the list or until sub list contains only one element.
- ❖ **Step 9 :** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

**Example**

Consider the following list of elements and search element :

|      |                |    |    |    |    |    |    |    |    |
|------|----------------|----|----|----|----|----|----|----|----|
|      | 0              | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10             | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |
|      | search element |    |    |    | 12 |    |    |    |    |

**Step 1 :** search element (12) is compared with middle element (50)

|      |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |
|      |    |    |    |    | 12 |    |    |    |    |

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20, & 32).

|      |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

**Step 2 :** search element (12) is compared with middle element (12)

|      |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |
|      |    |    |    |    | 12 |    |    |    |    |

Both are matching. So the result is "Element found at index 1".

**Example :**

search element 80

**Step 1 :** search element (80) is compared with middle element (50)

|      |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |
|      |    |    |    |    | 80 |    |    |    |    |



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

|      |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

**Step 2 :** search element (80) is compared with middle element (65)

|      |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |
|      |    |    |    |    |    |    |    | 80 |    |

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

|      |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

**Step 3 :** search element (80) is compared with middle element (80)

|      |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |
|      |    |    |    |    |    |    |    | 80 |    |

Both are matching. So the result is "Element found at index 7".

### Binary Search Program in C Programming Language

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int first, last, middle, size, i, sElement, list[100];
    clrscr();
    printf("Enter the size of the list:");
    scanf("%d", &size);
    printf("Enter %d integer values in Ascending order/n",size);
    for (i=0;i<size;i++)
        scanf("%d", &list[i]);
    printf("Enter value to be searched :");
    scanf("%d", &sElement);
    first = 0;
    Last = size-1;
    middle = (first+last)/2;
    while (first<=last)
    {
        if (list[middle] < sElement)
```



```

    first = middle + 1;
else if (list[middle] == sElement)
{
    printf("Element found at index %d.\n", middle);
    break;
}
else
    last = middle - 1;
    middle = (first + last) / 2;
}
if (first > last)
    printf("Element Not found in the list.");
    getch();
}

```

## Hashing

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives includes :

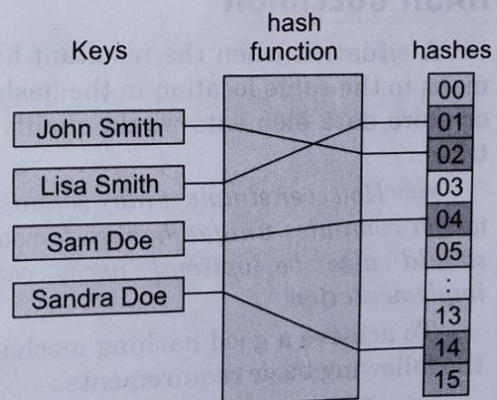
- ❖ In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- ❖ In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in  $O(1)$  time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

"It is a Data structure where the data elements are stored (inserted), searched, deleted based on the key generated for each element, which is obtained from a hashing function. In a hashing system the keys are stored in an array which is called the Hash Table. A perfectly





implemented hash table would always promise an average insert/delete/retrieval time of  $O(1)$ ."

Hashing is implemented in two steps :

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

$\text{hash} = \text{hashfunc}(\text{key})$

$\text{index} = \text{hash} \% \text{array-size}$

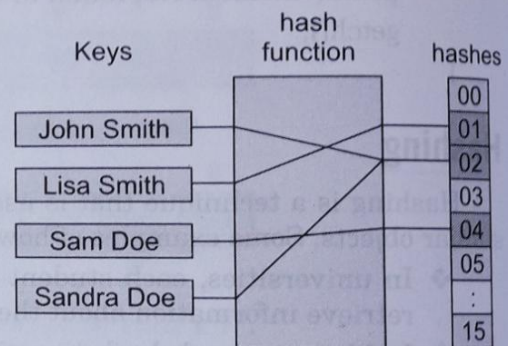
In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array-size - 1) by using the modulo operator (%).

### Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

A function which employs some algorithm to compute the key  $K$  for all the data elements in the set  $U$ , such that the key  $K$  which is of a fixed size. The same key  $K$  can be used to map data to a hash table and all the operations like insertion, deletion and searching should be possible. The values returned by a

**hash function** are also referred to as **hash** values, **hash** codes, **hash** sums, or **hashes**.



### HASH COLLISION

A situation when the resultant hashes for two or more data elements in the data set  $U$ , maps to the same location in the hash table, is called a hash collision. In such a situation two or more data elements would qualify to be stored/mapped to the same location in the hash table.

*==> However simple it may sound, it is practically never possible to find a hashing function which computes unique hashes for each element in the data set  $U$ . Further a hash function should also be optimal w.r.t. computing time and should offer adequate case of implementation.*

To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements :

1. **Easy to compute** : It should be easy to compute and must not become an algorithm in itself.
2. **Uniform distribution** : It should provide a uniform distribution across the hash table and should not result in clustering.
3. **Less collisions** : Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.



## HASH COLLISION RESOLUTION TECHNIQUES

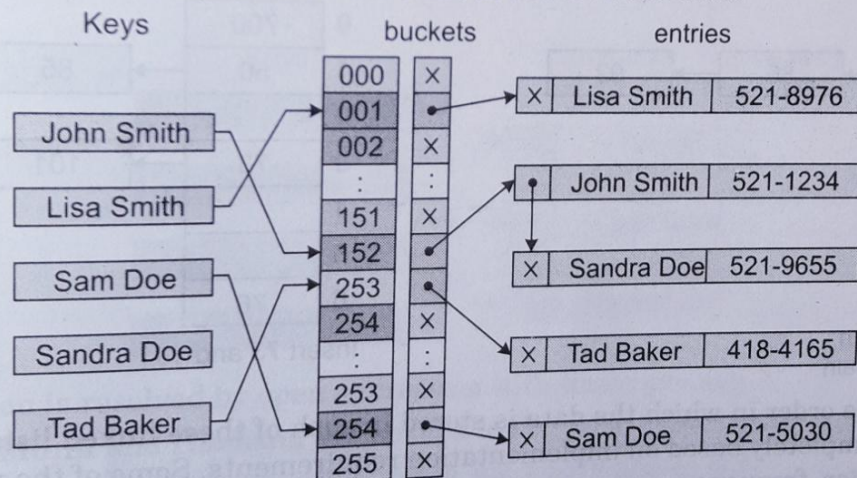
There are mainly two methods to handle collision :

1. Separate Chaining
2. Open Addressing

### Open Hashing (Separate chaining)

Open Hashing, is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.

In this technique when a data needs to be searched, it might become necessary (worst case) to traverse all the nodes in the linked list to retrieve the data.



### Another example :

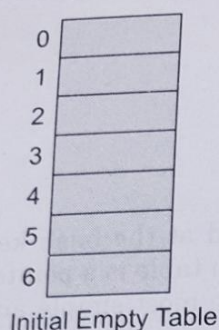
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

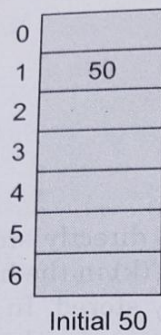
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

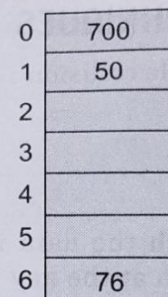




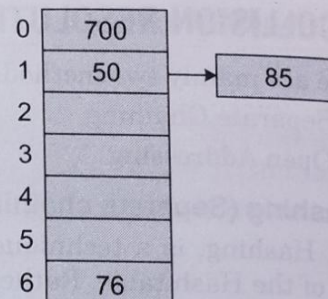
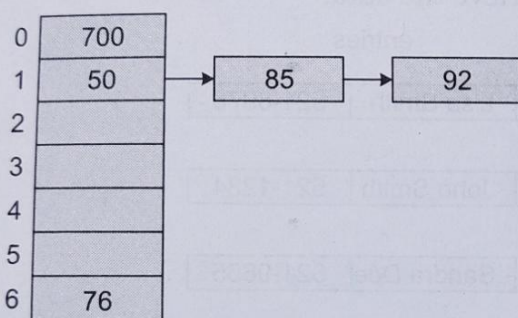
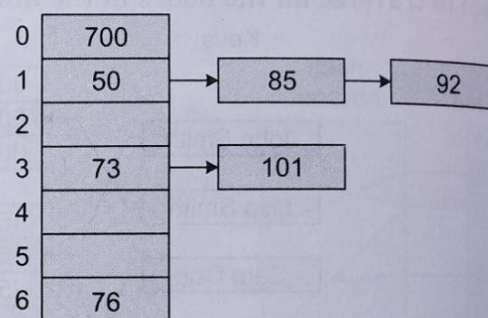
Initial Empty Table



Initial 50



Insert 700 and 76

Insert 85 : Collision  
Occurs, add to chainInsert 92 Collision  
Occurs, add to chain

Insert 73 and 101

Note that the order in which the data is stored in each of these linked lists (or other data structures) is completely based on implementation requirements. Some of the popular criteria are insertion order, frequency of access etc.

## CLOSED HASHING (OPEN ADDRESSING)

In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

The probe sequence is the sequence that is followed while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes.

When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table.

The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is **index**. The probing sequence for linear probing will be :

$$\text{index} = \text{index} \% \text{hash Table Size}$$



$\text{index} = (\text{index} + 1) \% \text{hash Table Size}$

$\text{index} = (\text{index} + 2) \% \text{hash Table Size}$

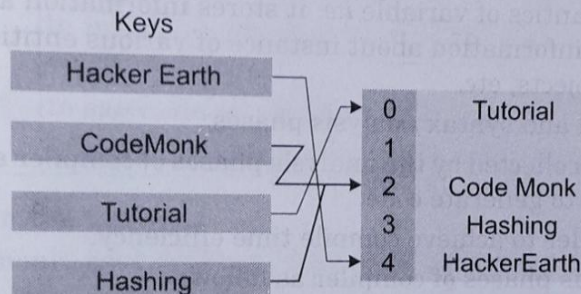
$\text{index} = (\text{index} + 3) \% \text{hash Table Size}$

and so on...

For which one of the following techniques is adopted :

1. Linear Probing (this is probe to clustering of data+ Some other constrains.
2. Quadratic probing.
3. Double hashing (in short in case of collision another hashing function is used with the key value as an input to identify wherein the open addressing scheme the data should actually be stored).

## Hash Table



Hash collision is resolved by open addressing with linear probing.

Since **CodeMonk** and **Hashing** are hashed to the same index i.e., 2,

store **Hashing** at 3 as the interval between successive probes is 1.

### Advantages :

1. Simple to implement.
2. Hash table never fills up, we can always add more elements to chain.
3. Less sensitive to the hash function or load factors.
4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

### Disadvantages :

1. Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
2. Wastage of Space (Some Parts of hash table are never used)
3. If the chain becomes long, then search time can become  $O(n)$  in worst case.
4. Uses extra space for links.

### A comparative analysis of Closed Hashing Vs Open Hashing

| Open Addressing                                                                                | Closed Addressing                                                         |
|------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| All elements would be stored in the Hash table itself. No additional data structure is needed. | Additional Data structure needs to be used to accommodate collision data. |



|                                                                                            |                                                                                            |
|--------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| In cases of collisions, a unique hash key must be obtained.                                | Simple and effective approach to collision resolution. Key may or may not be unique.       |
| Determining size of the hash table, adequate enough for storing all the data is difficult. | Performance deterioration of closed addressing much slower as compared to Open addressing. |
| State needs be maintained for the data (additional work)                                   | No state data needs to be maintained (easier to maintain)                                  |
| Uses space efficiently                                                                     | Expensive on space                                                                         |

### Symbol Table in Compiler

**Prerequisite :** Phases of a Compiler.

**Symbol Table** is an important data structure created and maintained by the compiler in order to keep track of semantics of variable *i.e.* it stores information about scope and binding information about names, information about instance of various entities such as variable and function names, classes, objects, etc.

- ❖ It is built in lexical and syntax analysis phases.
- ❖ The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.
- ❖ It is used by compiler to achieve compile time efficiency.
- ❖ It is used by various phases of compiler as follows :
  1. **Lexical Analysis** : Creates new table entries in the table, example like entries about token.
  2. **Syntax Analysis** : Adds information regarding attribute type, scope, dimension, line of reference, uses, etc. in the table.
  3. **Semantic Analysis** : Uses available information in the table to check for semantics *i.e.* to verify that expressions and assignments are semantically correct (type checking) and update it accordingly.
  4. **Intermediate Code generation** : Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
  5. **Code Optimization** : Uses information present in symbol table for machine dependent optimization.
  6. **Target Code generation** : Generates code by using address information of identifier present in the table.

**Symbol Table entries** : Each entry in symbol table is associated with attributes that support compiler in different phases.

#### Items stored in Symbol table :

- ❖ Variable names and constants
- ❖ Procedure and function names
- ❖ Literal constants and strings
- ❖ Compiler generated temporaries
- ❖ Labels in source languages

#### Information used by compiler from Symbol table :



- ❖ Data types and name
- ❖ Declaring procedures
- ❖ Offset in storage
- ❖ If structure or record, then pointer to structure table.
- ❖ For parameters, whether parameter passing by value or by reference
- ❖ Number and type of arguments passed to function
- ❖ Base Address.

**Operations of Symbol table :** The basic operations defined on a symbol table include :

| Operation     | Function                                                             |
|---------------|----------------------------------------------------------------------|
| allocate      | to allocate a new empty symbol table                                 |
| free          | to remove all entries and free storage of symbol table               |
| lookup        | to search for a name and return pointer to its entry                 |
| insert        | to insert a name in a symbol table and return a pointer to its entry |
| set_attribute | to associate an attribute with a given entry                         |
| get_attribute | to get an attribute associated with a given entry                    |

### Implementation of Symbol table

Following are commonly used data structure for implementing symbol table :

#### 1. List :

- ❖ In this method, an array is used to store names and associated information.
- ❖ A pointer "**available**" is maintained at end of all stored records and new names are added in the order as they arrive.
- ❖ To search for a name we start from beginning of list till available pointer and if not found we get an error "**use of undeclared name**".
- ❖ While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. "**Multiple defined name**".
- ❖ Insertion is fast  $O(1)$ , but lookup is slow for large tables— $O(n)$  on average
- ❖ Advantage is that it takes minimum amount of space.

#### 2. Linked List :

- ❖ This implementation is using linked list. A link field is added to each record.
- ❖ Searching of names is done in order pointed by link of link field.
- ❖ A pointer "**First**" is maintained to point to first record of symbol table.
- ❖ Insertion is fast  $O(1)$ , but lookup is slow for large tables— $O(n)$  on average.

#### 3. Hash Table :

- ❖ In hashing scheme two tables are maintained : a hash table and symbol table and is the most commonly used method to implement symbol tables.
- ❖ A hash table is an array with index range : 0 to  $\text{tablesize}-1$ . These entries are pointer pointing to names of symbol table.
- ❖ To search for a name we use hash function that will result in any integer between 0 to  $\text{tablesize}-1$ .



- ❖ Insertion and lookup can be made very fast— $O(1)$ .
- ❖ Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.

#### 4. Binary Search Tree :

- ❖ Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
- ❖ All names are created as child of root node that always follow the property of binary search tree.
- ❖ Insertion and lookup are  $O(\log_2 n)$  on average.

### Operations

A symbol table, either linear or hash, should provide the following operations :

#### Insert () :

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert () function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example :

int a;

should be processed by the compiler as :

insert(a, int) :

#### lookup() :

lookup () operation is used to search a name in the symbol table to determine :

- ❖ if the symbol exists in the table.
- ❖ if it is declared before it is being used.
- ❖ if the name is used in the scope.
- ❖ if the symbol is initialized.
- ❖ if the symbol declared multiple times.

The format of lookup () function varies according to the programming language. The basic format should match the following :

#### lookup (symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.



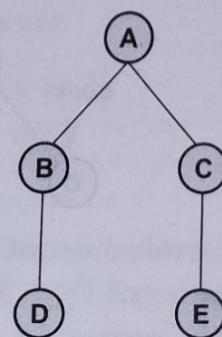
प्रश्न 1. Tree क्या होता है? ये कितने प्रकार के होते हैं?

उत्तर—Tree कई सारे नोड्स (Nodes) का Collection होता है जिसमें Nodes एक-दूसरे से Directed या Non-directed edges के द्वारा एक-दूसरे से जुड़े होते हैं। इसके द्वारा किसी data item के बीच के रिलेशन को Hierarchical (parent-child) के form में represent कर सकते हैं।

Tree एक Non-linear data structure है। Tree में पहला element Root node कहलाता है और remaining elements Non-empty sets के रूप में बट जाते हैं जो कि उस Root का Sub-tree कहलाता है।

Natural tree Ground से ऊपर की ओर बढ़ता है जबकि Tree Data Structure ऊपर से नीचे की ओर बढ़ता है।

दिये गये Picture में, A Root node nw Deewj B, C sub-tree nw root node A के।



**Tree के प्रकार :**

1. General Tree

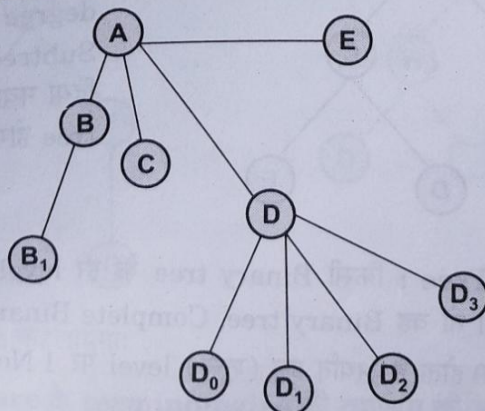
2. Binary Tree

3. Strictly Binary tree,

4. Complete Binary tree,

5. Extended Binary tree.

1. **General Tree :** General Tree में कितने भी No. of Subtree हो सकते हैं। यहाँ पर A root node है जबकि B, C, D और E Root Node A के Subtrees हैं।



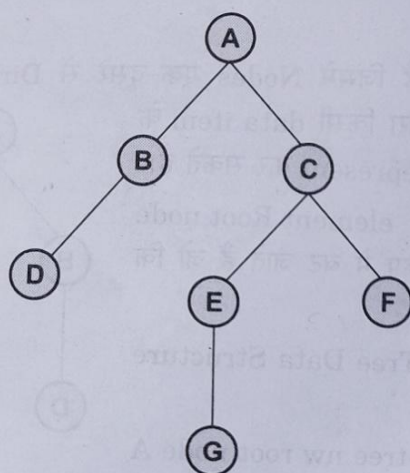


**2. Binary Tree :** Binary tree भी एक Non-linear data Structure है जिसमें किसी Node की Degree 0(शून्य), 1(एक) या अधिक से अधिक दो (Left subtree और Right subtree) होते हैं, Binary tree कहलाता है।

नोट : (1) किसी भी Level पर किसी भी node की degree दो से ज्यादा नहीं होनी चाहिए।

(2) किसी भी Node की Degree दो से अधिक नहीं हो सकती है। Node की Degree 0, 1, 2 maximum हो सकती है।

**Example :** यहाँ पर A Root Node है और इसकी Degree 2 (B,C) है। सारे Node की Degree इस प्रकार है—

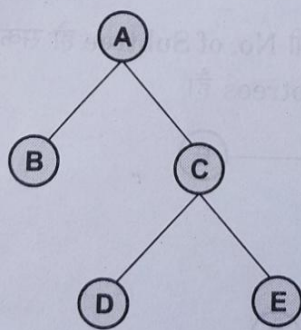


| Node | Degree |
|------|--------|
| A    | 2      |
| B    | 1      |
| C    | 2      |
| D    | 0      |
| E    | 1      |
| F    | 0      |
| G    | 0      |

यहाँ पर किसी भी Node की Degree दो से ज्यादा नहीं है इसलिए यह tree Binary Tree है।

**3. Strictly Binary Tree :** किसी Binary tree में जितने भी Non-terminal nodes हैं उनकी degree या तो शून्य (0) है या दो, या किसी terminal node की Left subtree और right subtree दोनों हो तो इस Binary tree को strictly Binary tree कहेंगे।

**Example :**



यहाँ पर A Root node है और इसकी B (Left) और C right subtree है। अर्थात् degree दो हैं फिर Root node है। Subtree में जिनकी degree दो है इस प्रकार दिया गया Binary tree, strictly Binary tree होगा।

**4. Complete Binary Tree :** किसी Binary tree के हर level पर data item (Node)  $2^n$  सूत्र (formula) के अनुसार हो तो वह Binary tree, Complete Binary tree कहलाता है। जैसे कि—  
 $2^n = 2^0 = 1$ , Root का level 0 होता है अर्थात् इस (शून्य) level पर 1 Node होगा, जहाँ  $n = \text{level number}$  है उस Tree का।

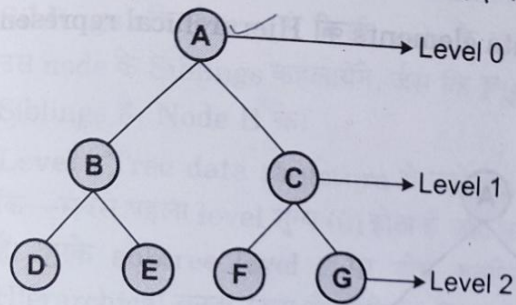
$2^1 = 2$ , level 1 पर 2 Node होंगे।



$2^2 = 4$ , level 2 पर 4 Node होंगे।

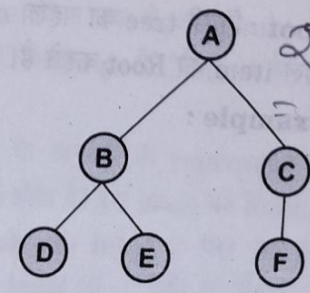
$2^3 = 8$ , level 3 पर 8 Node होंगे।

इसी प्रकार हर level पर कितने node होने चाहिए वह इस सूत्र ( $2^n$ ) पर निर्भर करेगा।



Complete binary tree

Figure 1 में सारे Nodes  $2^n$  सूत्र के अनुसार हैं।



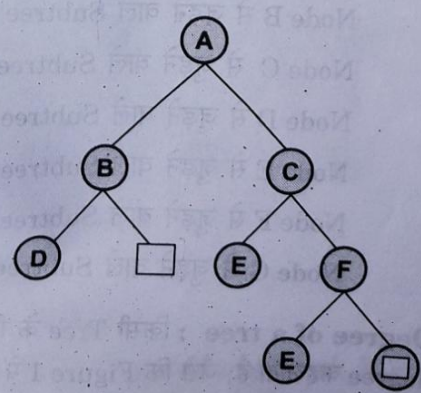
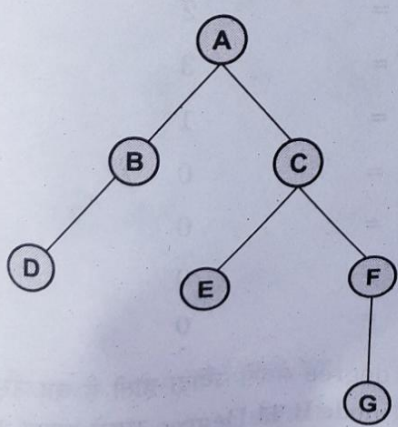
Binary tree

जबकि Figure 2 में Node  $2^n$  के अनुसार नहीं हैं।

### Extended Binary Tree

एक Binary tree, Extended Binary tree तब कहलाता है यदि हर node जिसकी degree (subtree) 0 (शून्य) हो या तो degree (subtree) 2 हो। इस Tree में जिस subtree की degree 0 होती है उसको External Node और जिसकी degree दो होती है उसे Internal Node कहते हैं। External Node को squares और internal node को circles के रूप में represent करते हैं।

जैसे कि यहाँ पर एक tree लिया गया है इसमें सारे Node internal हैं इस tree को Extended Binary नहीं कहेंगे। Extended Binary tree में convert करने के लिए जिस node की degree 1 है उसमें external node add करेंगे जिसको rectangle से represent किया गया है।



तब Extended Binary tree कहलायेगा।

प्रश्न 2. Tree data structure के terminologies की व्याख्या कीजिए।

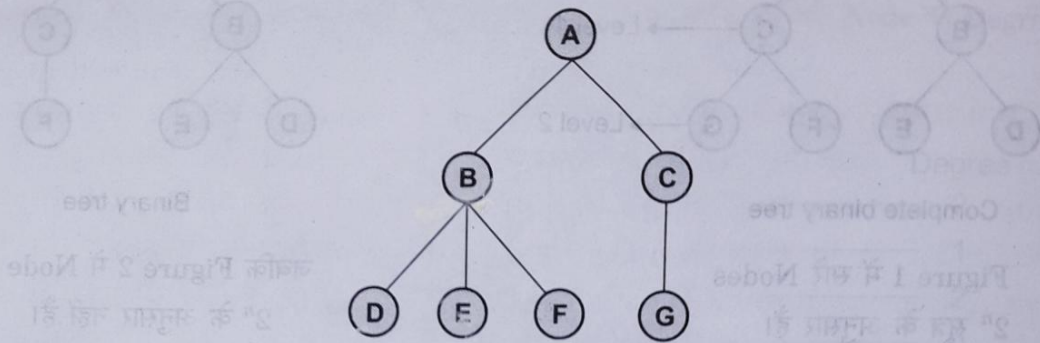
उत्तर—Tree data structure के terminologies निम्न हैं—



- |                  |                      |                     |                     |
|------------------|----------------------|---------------------|---------------------|
| 1. Root          | 2. Node              | 3. Degree of a Node | 4. Degree of a tree |
| 5. Terminal node | 6. Non-terminal node | 7. Siblings         |                     |
| 8. Level         | 9. Edge              | 10. Path            | 11. Depth           |
| 12. Forest.      |                      |                     |                     |

1. **Root** : किसी tree का पहला node या किसी data elements की Hierarchical representation में पहले item को Root कहते हैं।

**Example :**



इस Hierarchical representation में A सबसे पहला node (data item) है इसलिए इस Node को Root node कहते हैं।

2. **Node** : Tree Data structure में किसी data item को store करने के लिए node का प्रयोग किया जाता है। जैसे ऊपर दिए गये चित्र में A, B, C, D, E, F, G कुल सात (7) Nodes हैं।
3. **Degree of a Node** : किसी Tree के एक Node से जितने subnode या subtree जुड़े होते हैं वह इस node की degree कहलाता है, जैसे कि figure 1 में।

| Node या Subnode               |           |   | Degree |
|-------------------------------|-----------|---|--------|
| Node A से जुड़ने वाले Subtree | = B, C    | = | 2      |
| Node B से जुड़ने वाले Subtree | = D, E, F | = | 3      |
| Node C से जुड़ने वाले Subtree | = G       | = | 1      |
| Node D से जुड़ने वाले Subtree | = Nil     | = | 0      |
| Node E से जुड़ने वाले Subtree | = Nil     | = | 0      |
| Node F से जुड़ने वाले Subtree | = Nil     | = | 0      |
| Node G से जुड़ने वाले Subtree | = Nil     | = | 0      |

4. **Degree of a tree** : किसी Tree के जिस Node की degree सबसे ज्यादा होती है वह उस tree की degree कहलाता है, जैसे कि Figure 1 में 7 Nodes हैं पर node B की Degree सबसे ज्यादा 3 है इसलिए इस tree की degree 3 होगी।
5. **Terminal Node** : किसी tree के उस node को terminal node कहते हैं जिसकी degree शून्य (0) होती है। जैसे Figure 1 में D, E, F और G ऐसे Nodes हैं जिनकी कोई और subtree नहीं है या जिनकी degree शून्य है।



6. **Non-terminal Node :** Tree के ऐसे Node जिसकी degree शून्य (0) न हो या जिसकी कोई न कोई subtree अवश्य हो, ऐसे node को Non-terminal node कहते हैं। जैसे कि Fig. 1 में A, B, C Non-terminal nodes हैं क्योंकि इनकी Degree एक या एक से ज्यादा है।
7. **Siblings :** किसी tree में किसी node के एक से ज्यादा degree हों या एक से ज्यादा subtree हों तो वह उस node के Siblings कहलायेंगे, जैसे कि Figure 1 में B और C Siblings हैं, Node A के। D, E और F Siblings हैं, Node B के।
8. **Level :** Tree data structure में सारे data items, level के अनुसार ही represent होते हैं, जैसे कि—सबसे पहला level शून्य (0) होता है जहाँ पर केवल एक node होता है। इस node को Root node कहते हैं। इसके subtree level 1 पर होंगे जबकि level 1 के subtree level 2 पर। इस प्रकार tree Hierarchical बनता रहता है। जैसे कि दिए गये Fig. 1 में कुल 3 level (0, 1, और 2) हैं।
9. **Edge :** किसी Tree के किन्हीं 2 nodes को जोड़ने के लिए एक सीधी लाइन का प्रयोग करते हैं। इस लाइन को ही edge कहते हैं। यह लाइन, directional और non-directional दोनों प्रकार की हो सकती हैं।
10. **Path :** किसी Tree को एक node से दूसरे node पर जाने के लिए जिन edge का प्रयोग करते हैं उन edges sequential collection को Path कहते हैं, जैसे कि fig. 1 में A से F तक जाना है, तब पहले (A से B) और फिर (B से F) तक जाएंगे।  
सफेद → (A से B), (B से F)
11. **Depth :** किसी Tree का जितना level होता है वह level उस tree का Depth कहलाता है। इसी को उस tree की height भी कहा जाता है, जैसे—दिए गये figure 1 में tree की depth 2 है।
12. **Forest :** किसी दिए गये tree को root node से अलग करने पर tree subtree में बट जाता है तथा उस subtree और subtrees में बाँटा जाता है। इस प्रकार वह tree, forest में बदल जाता है।

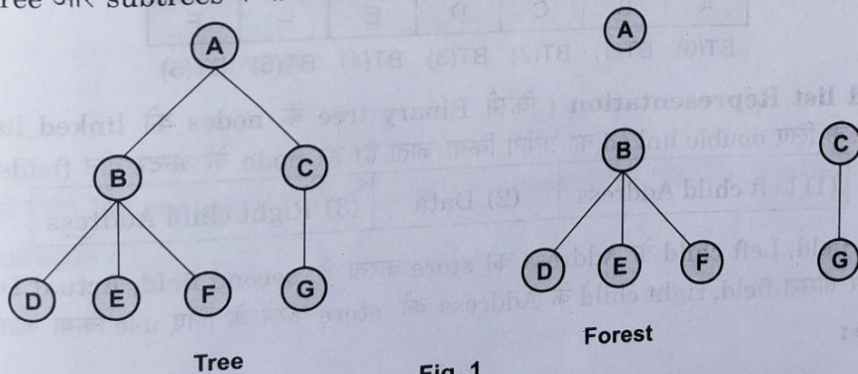


Fig. 1

प्रश्न 3. Binary tree को कितने प्रकार से represent कर सकते हैं?

उत्तर—Binary tree data structure को दो methods के द्वारा represent कर सकते हैं। ये methods निम्न प्रकार के हैं—

1. Array Representation

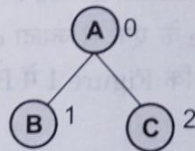
2. Linked list Representation

**1. Array Representation :** Binary tree के nodes को store करने के लिए array का प्रयोग किया जाता है। Array में store हर node को एक के बाद एक (sequential) access किया जाता है। 'C' language में array का index 0 (शून्य) से शुरू होता है और (maxsize-1) तक होता है। यहाँ Binary tree के node भी 0



यहाँ से शुरू होकर (Maxsize-1) तक represent किया जाता है। इसमें Root node हमेशा Index 0 पर होता है। इसके बाद के node, left child और right child के रूप में store होंगे।

**Example :**



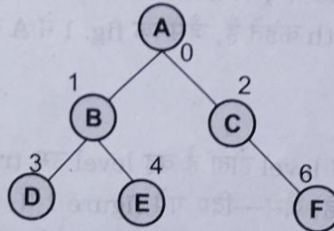
Array के द्वारा इस Binary tree (BT) को निम्न प्रकार से represent करते हैं:

|   |   |   |
|---|---|---|
| A | B | C |
|---|---|---|

BT(0) BT(1) BT(2)

यहाँ पर A, B और C का parent node (Root node) है। B, Root node A का left child है और C, Root node A का right child है।

**Example :**



इस tree को Array में निम्न प्रकार से represent करते हैं—

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| A | B | C | D | E | - | F |
|---|---|---|---|---|---|---|

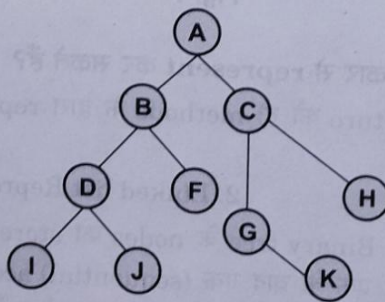
BT(0) BT(1) BT(2) BT(3) BT(4) BT(5) BT(6)

**2. Linked list Representation :** किसी Binary tree के nodes को linked list के रूप में represent करने के लिए double linked का प्रयोग किया जाता है। हर node के अन्दर तीन fields होते हैं।

|                        |          |                         |
|------------------------|----------|-------------------------|
| (1) Left child Address | (2) Data | (3) Right child Address |
|------------------------|----------|-------------------------|

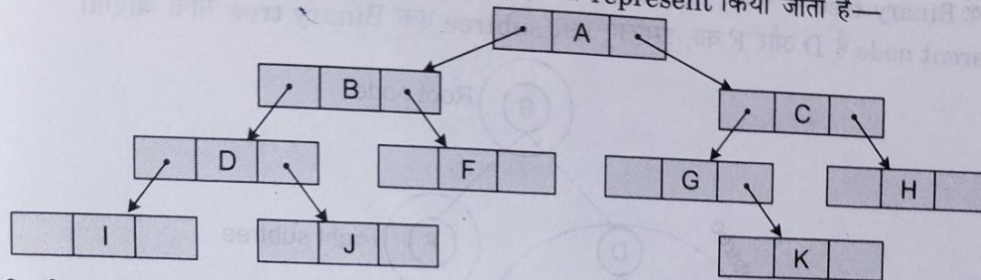
Node का पहला field, Left child के Address को store करता है, second field, Actual Data को store करने के लिए और तीसरा field, right child के Address को store करने के लिए use किया जाता है।

**Example :**





इस Binary tree को linked list के द्वारा निम्न प्रकार से represent किया जाता है—



अगर किसी Binary tree को linked list के द्वारा represent करना हो तो ऊपर दिये गये तरीके का ही प्रयोग करना होगा।

#### प्रश्न 4. Binary tree traversal क्या होता है? ये कितने प्रकार के होते हैं?

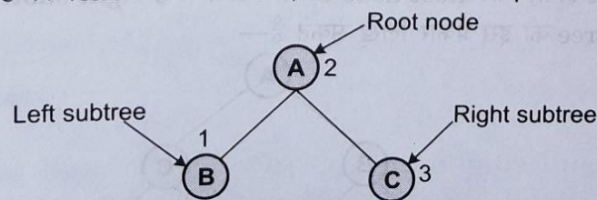
उत्तर—Tree traversal, tree data structure का एक common operation है जिसके द्वारा tree के सभी elements को कम से कम एक बार जरूर visit करना होगा (इसमें सारे nodes जरूर display होने चाहिए)। Binary tree के node किस प्रकार display होंगे यह traversal method पर निर्भर करता है।

नोट—Binary tree के सभी nodes को display करना या visit करना, Binary tree traversal कहलाता है। यहाँ पर तीन प्रकार के Binary tree traversals होते हैं—

1. In-order traversal
2. Pre-order traversal
3. Post-order traversal

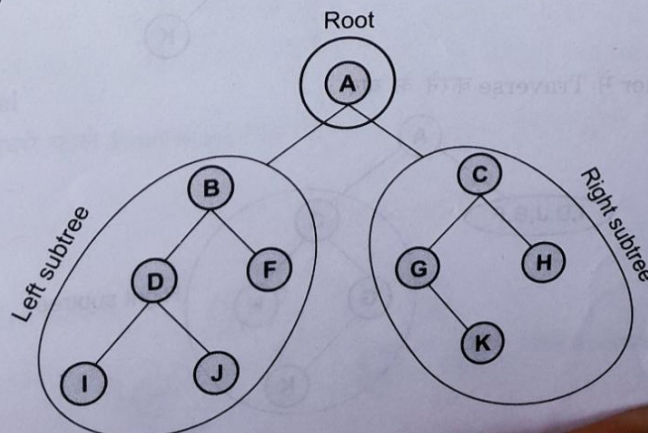
इस प्रकार के traversal, non-empty binary tree में ही किया जा सकता है।

**1. In-order Traversal :** इस प्रकार के Traversal में सबसे पहले tree का left subtree, फिर Root और अन्त में right subtree को visit किया जाता है। इसमें हर node स्वयं एक subtree होता है। जैसे कि—



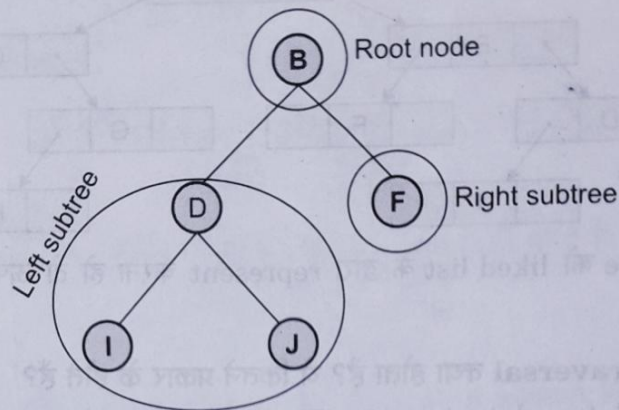
इसमें सबसे पहले left subtree B visit होगा, फिर root node A और अन्त में right subtree C visit होगा।

**दूसरा Example :**

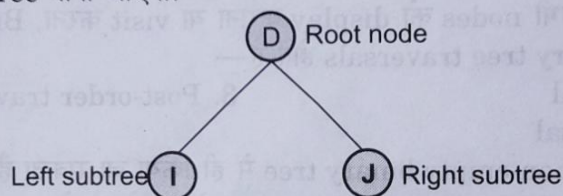




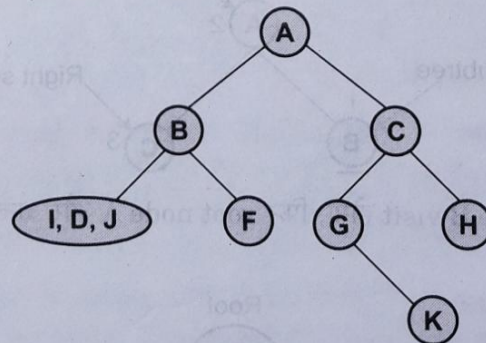
दिए गये Binary tree में सबसे पहले left subtree को visit करते हैं पर left subtree का सबसे पहला node B Parent node है D और F का, इसलिए left subtree एक Binary tree माना जाएगा।



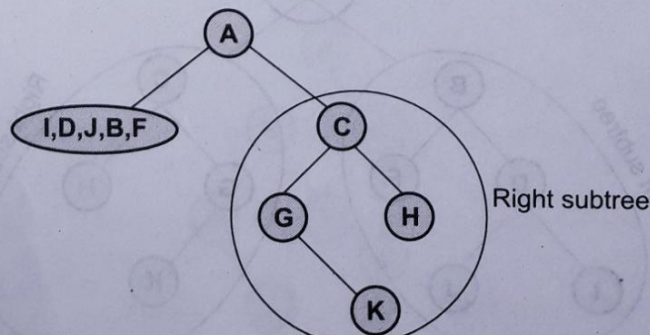
इस tree में सबसे पहले left subtree D पर visit किया जाएगा पर D parent node है I और J का, इसलिए left subtree, एक Binary tree माना जाएगा।



अब सबसे पहले I visit होगा, फिर Root node D और अन्त में J right subtree visit होगा। तब visited nodes होंगे (I, D, J)। तब tree को इस प्रकार लिख सकते हैं—

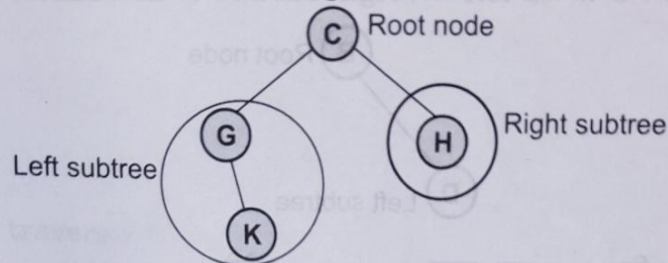


इस tree को In-order में Traverse करने के बाद

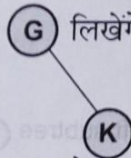




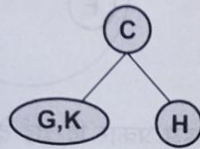
अब left subtree को Traverse करने के बाद Right subtree को इस प्रकार लिखेंगे—



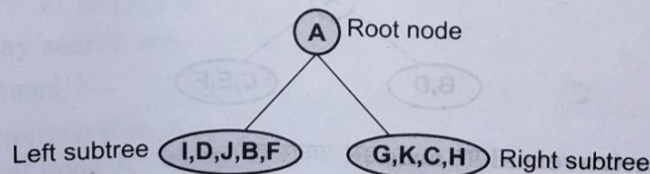
Left subtree का node G root node है, इसलिए इसको इस प्रकार लिखेंगे।



अब traverse करने पर G का left subtree नहीं है तब Root node G को traverse करेंगे, फिर K right subtree को। अब tree को इस प्रकार लिखेंगे—



फिर



### In-order Traversal :

I, D, J, B, F    A    G, K, C, H  
left subtree   'Root'   right subtree

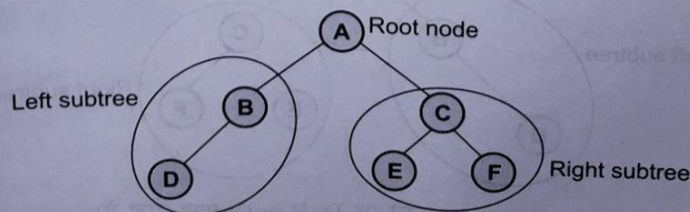
#### In-order Algorithm :

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

### Pre-order Traversal

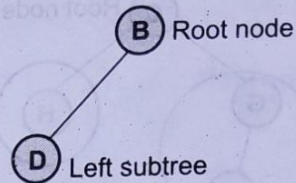
इस traverse में सबसे पहले Root node, फिर left subtree और अन्त में right subtree.

**Example :**

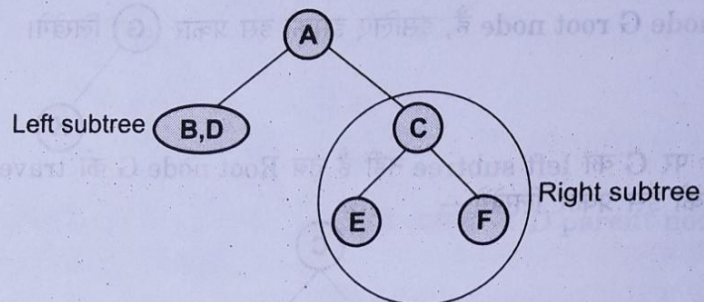




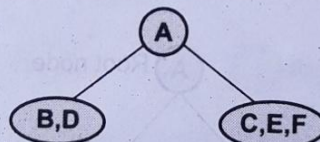
Root node A का B और C क्रमशः left और right subtree हैं। Left subtree को traverse करने पर



B, D nodes प्राप्त होंगे, फिर tree इस प्रकार represent करेंगे—



Right subtree traverse करने पर tree इस प्रकार दिखाई देगा—



इस प्रकार **Pre-order traversal** पर node इस प्रकार होंगे—

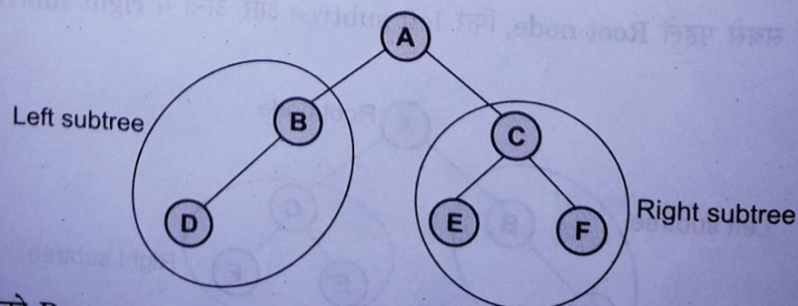
A, B, D, C, E, F

### Pre-order Algorithm

1. Visit the root (सबसे पहले root node visit करते हैं)
2. Traverse the left subtree (फिर left subtree visit करेंगे)
3. Traverse the right subtree (और अन्त में right subtree visit करेंगे)

### Post-order Traversal

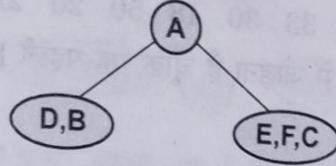
इसमें सबसे पहले Left subtree, फिर right subtree और अन्त में root node visit होता है। जैसे कि—



Left subtree को Post-order traverse करने पर D, B path प्राप्त होता है।



Right subtree को Post-order में traverse करने पर E, F, C पाथ प्राप्त होता है। इसे इस प्रकार लिखेंगे—



इस प्रकार Post-order traversal में node इस प्रकार visit होंगे—

D, B, E, F, C, A

### Algorithm post-order Traverse :

1. Traverse the left subtree (पहले left subtree visit करेंगे)
2. Traverse the right subtree (फिर right subtree visit करेंगे)
3. Visit root node. (और अन्त में Root node visit करेंगे)

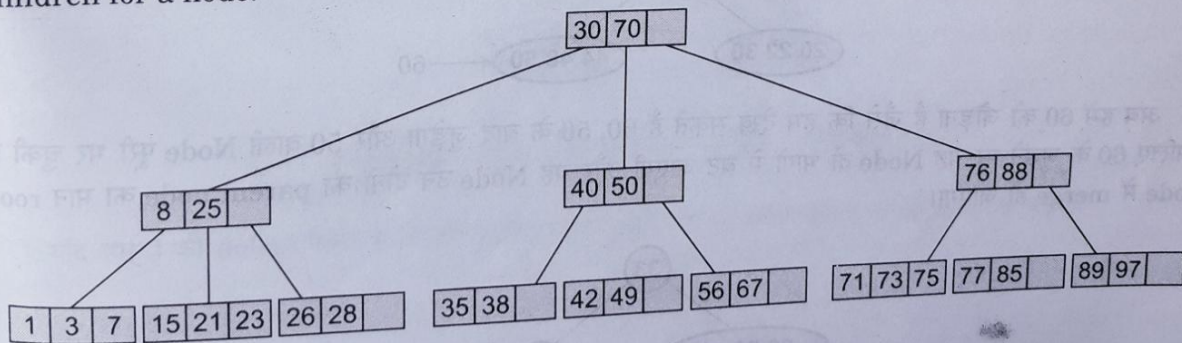
### प्रश्न 5. B-Tree पर संक्षिप्त टिप्पणी कीजिए।

**उत्तर—B-Tree :** B-Tree को balanced tree या balanced M-way tree भी कहा जाता है। Binary search tree, AVL tree, Red-black tree इत्यादि सभी की केवल एक key value होती है और maximum दो nodes (children) होते हैं जबकि B-Tree में node एक से ज्यादा Key value store होता है और इसके दो से ज्यादा (children) node होते हैं। B-Tree का निर्माण 1972 में Bayer और Mc Creight ने किया था जिसका नाम Height Balanced M-way search tree रखा गया। बाद में इसका नाम B-Tree कर दिया गया।

**B-Tree** के निम्न विशेषताएँ हैं—

1. सभी Leaf (Terminal) nodes एक ही level पर होने चाहिए।
2. सभी node में, root node को छोड़कर कम से कम  $[m/2 - 1]$  keys और ज्यादा से ज्यादा  $m - 1$  keys होंगे।
3. सभी non-leaf node, root node को छोड़कर की कम से कम  $m/2$  child होंगे।
4. Node के सभी key values, Ascending order में होने चाहिए।

**Example :** B-Tree of order 4 contains maximum 3 key values in a node and maximum 4 children for a node.



### प्रश्न 6. B-Tree पर होने वाले operations की व्याख्या कीजिए।

**उत्तर—1. Addition :** B-Tree में किसी भी key को जोड़ने से पहले यह देखा जाता है कि वह key कहाँ जोड़ी जानी है। यदि key पहले से निर्मित node में जा सकती है तो key को जोड़ना आसान है। वह node को दो भागों में बाँट देती है। वह key जिसका मान बीच का होता है, वह Node में रहता है और इसके बायें child में इस key के दायें स्थित समस्त keys आ जाती हैं।



**Example :** 4 order को एक Tree बनाना है।

44 33 30 48 50 20 22 60

सबसे पहले key 44 है जिसे हमें tree में जोड़ना है चूँकि यह पहली key है। इसलिए इस key के द्वारा ही Root node का निर्माण होगा :

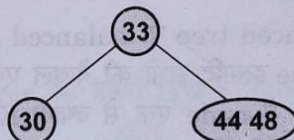
44

इसी प्रकार हम 33 और 30 को भी उनके स्थान के अनुरूप node में जोड़ देंगे।

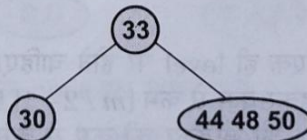
44 33 30

हमें यह tree 4 order का बनाना है इसलिए इसके एक node में अधिकतम 3 keys ही रख सकते हैं। अतः अब जो भी key इसमें जोड़ी जाएगी वह इस (root) node को दो भागों में बाँट देगी। यहाँ हमें 48 जोड़ना है।

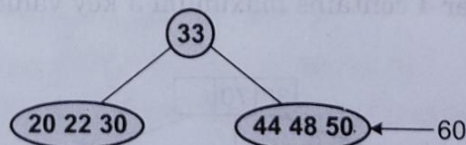
44 33 30 48



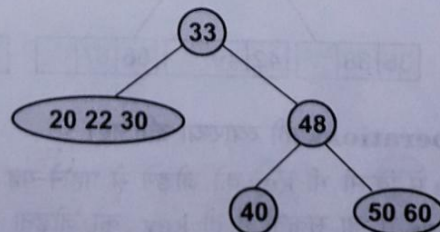
अब हम 50 को इसके स्थान के अनुरूप जोड़ देंगे।



अब हम 20 और 22 को इनके स्थान के अनुरूप जोड़ देंगे।

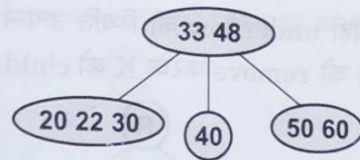


अब हमें 60 को जोड़ना है जैसे कि हम देख सकते हैं 60, 50 के बाद जुड़ेगा और 50 वाली Node पूरी भर चुकी है इसलिए 60 के जुड़ने पर यह Node दो भागों में बट जाएगी और यह Node उन दोनों का parent node का मान root node में merge हो जाएगा।



Final tree इस प्रकार बनेगा—



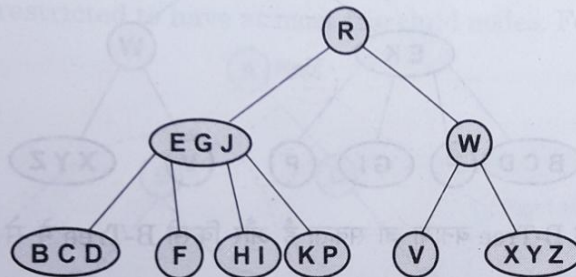


अब यह tree Balanced है और इस प्रकार बनाया गया है कि इसकी समस्त leaves एक ही level पर हैं।

**2. Deletion :** जैसा कि हम जोड़ने में पहले key तथा उसका स्थान देखते हैं, इसी प्रकार delete करने में भी पहले key तथा इसके स्थान को ढूँढ़ते हैं।

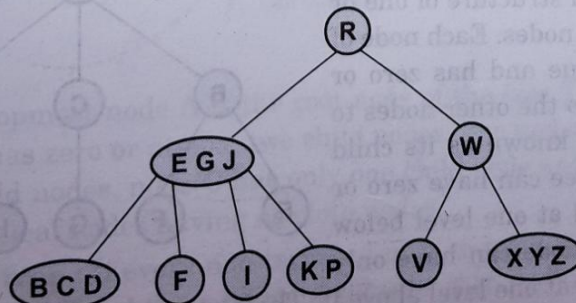
यदि delete की जाने वाली key एक Terminal (leaf) है जिसको delete करना सरल है अर्थात् केवल उस key को node में से मिटा देना होगा।

यदि delete की जाने वाली key एक terminal (leaf) नहीं है तो इसे इसके successor के मान से बदल दिया जाता है। इस प्रकार यदि successor, terminal node में है तो वहाँ से वह delete कर दिया जाएगा।

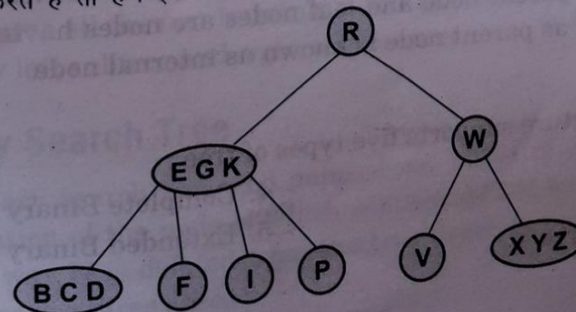


इस प्रकार यदि successor, non-terminal node में है तो इसका successor, replace होगा और यही सारी conditions इस successor के साथ भी लगेंगी।

यदि हम 'H' को delete करते हैं तो बहुत आसान है अर्थात् केवल 'H', node में से delete कर देंगे।

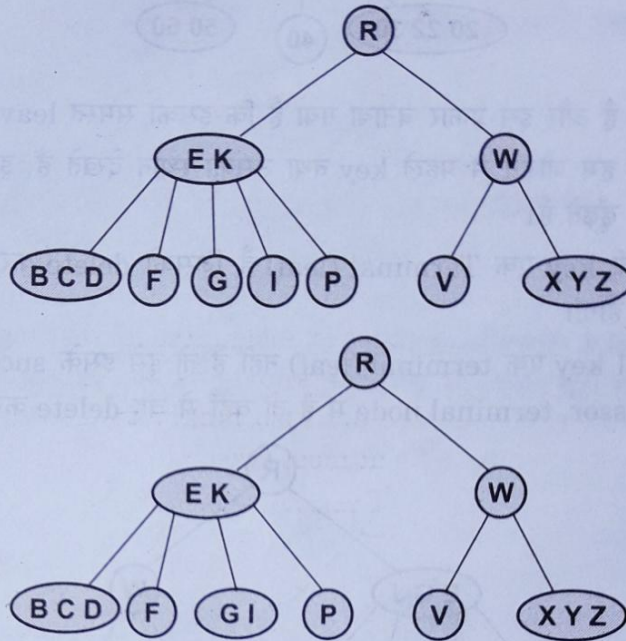


यदि हम J को delete करते हैं तो हमें इसके successor, K को इसके स्थान पर कॉपी करना पड़ेगा।





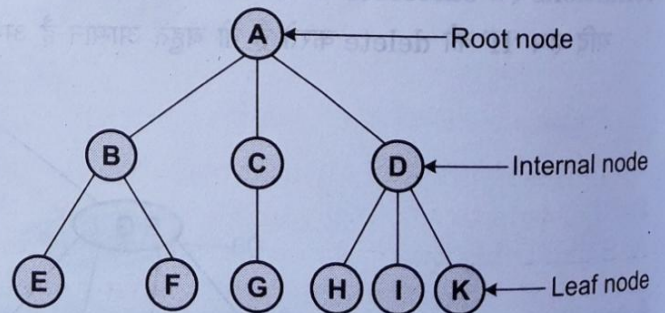
यदि हम 'F' को delete करते हैं तो यहाँ underflow की स्थिति उत्पन्न हो जाएगी। यहाँ L के delete होते ही G का एक child delete हो जाएगा। अतः G को remove करके K को child node में भेज दिया जाएगा।



इस प्रकार दिए गये क्रम का B-Tree बनाया जा सकता है और किसी B-Tree में से key को delete भी किया जा सकता है।

## Tree

A tree is a non-linear Data structure representing hierarchical structure of one or more elements known as nodes. Each node of a tree stores a data value and has zero or more Pointers pointing to the other nodes to the tree, which are also known as its child nodes. Each node in a tree can have zero or more child nodes located at one level below it. However, each child node can have only one parent node which is at one level above it. The node at the top of the tree is known as the root of the tree and the nodes at the lowest level are known as the leaf nodes. The root node is a special node having no parent node and leaf nodes are nodes having no child node. A node having child node as well as parent node is known as internal node.



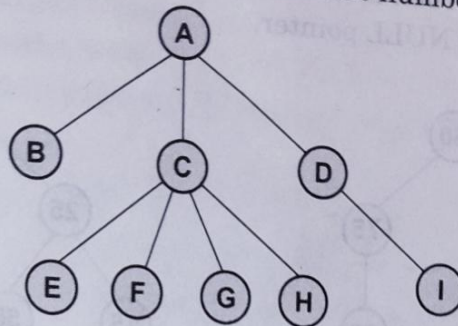
## Types of Tree

Non-linear data structure supports five types of tree :

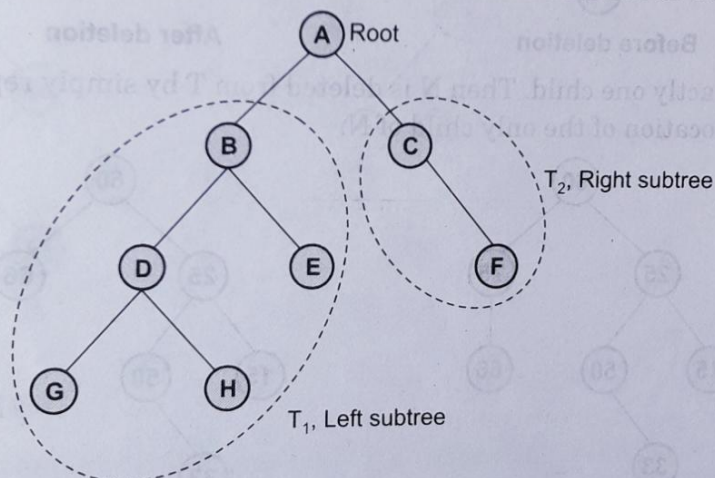
1. General tree
2. Binary tree
3. Strictly Binary tree
4. Complete Binary tree
5. Extended Binary tree



**1. General tree :** A general tree consists of various numbers of subtrees :



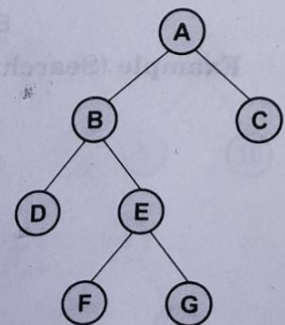
**2. Binary tree :** A binary tree is a special type of tree which can be either empty or has finite set of nodes, such that one of the nodes is designated as root node and remaining nodes are partitioned into two subtrees of root node known as left subtree and right subtree. The non-empty left subtree and right subtree are also binary tree. Unlike a general tree, each node in a binary tree is restricted to have at most two child nodes. For example,



In this figure, the topmost node A is the root node of the tree T. Each node in this tree has zero or atmost two child nodes. The nodes A, B and D has two child nodes, node C has only one child node and nodes G, H, E and F are leaf nodes having no child nodes.

**3. Strictly Binary tree :** If every non-terminal node in a binary tree consists of non-empty left subtree and right subtree, then such a tree is called strictly Binary tree. For example;

In the above binary tree all the non-terminal nodes such as B and G are having non-empty left subtree and right subtree tree.



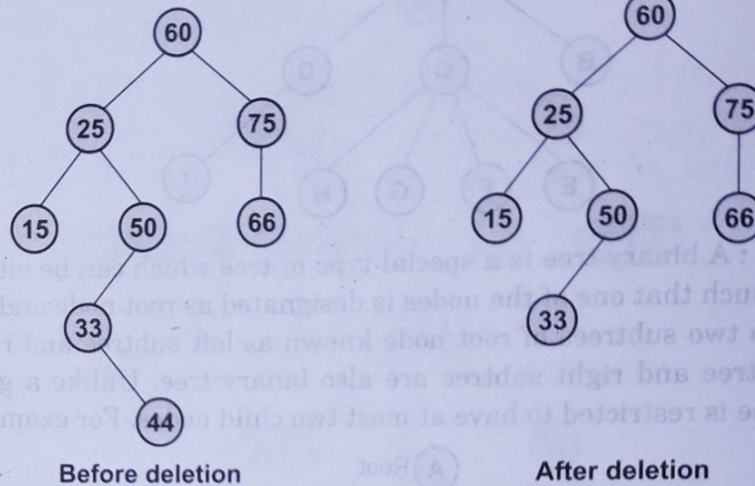
## Deleting in a Binary Search Tree

Suppose T is a binary search tree and suppose an ITEM of information is given. For deletion to find the location of the node N which contains ITEM and also the location of the parent node P(N). The way N is deleted from the tree depends primarily on the number of children of node N. There are three cases :

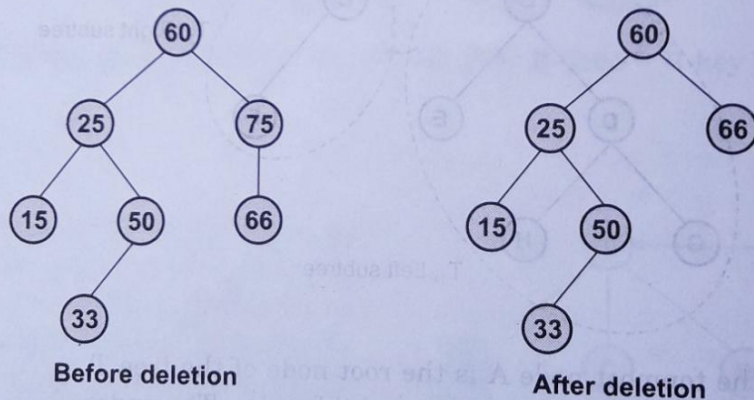


**Case 1.** N has no children. Then N is deleted from T by simply replacing the location of N in the Parent node P (N) by the NULL pointer.

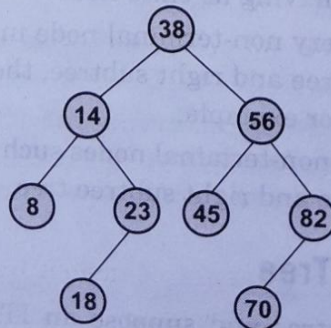
**Example :**



**Case 2.** N has exactly one child. Then N is deleted from T by simply replacing the location of N in P (N) by the location of the only child of N.



**Example (Searching) :**



Consider the binary search tree T. Suppose ITEM=18 is given. We get the following steps:

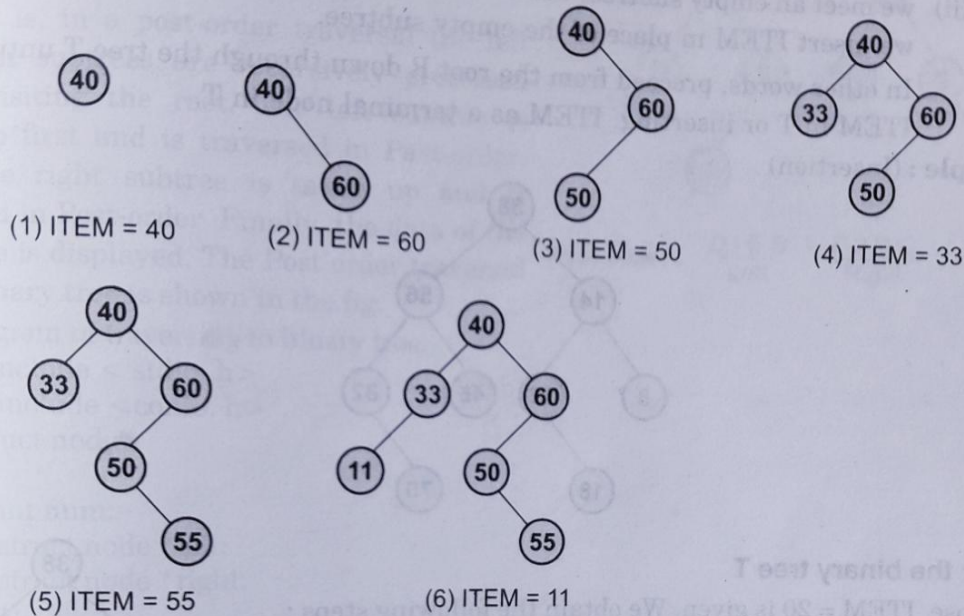
1. Compare ITEM = 18 with the root of the tree, 38. Since  $18 < 38$  proceed to the left child of 38, which is 14.



2. Compare ITEM = 18 with 14. Since  $18 > 14$ , proceed to the right child of 14, which is 23.
3. Compare ITEM = 18 with 23. Since  $18 < 23$ , proceed to the left child of 23, which is 18.
4. Compare ITEM = 18 with the left child, 18. We have found the location of 18 in the tree.

**Example :** Suppose the following six (6) numbers are inserted in order into an empty binary search tree.

40, 60, 50, 33, 55, 11

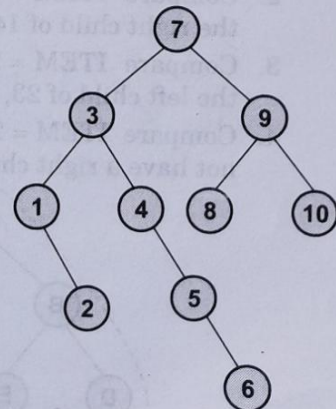


## Binary Search Tree

A binary search tree is a binary tree which is either empty or satisfies the following rules :

1. The value of the key in the left child or left subtree is less than the value of the root.
2. The value of the key in the right child or right subtree is more than or equal to the value of the root.
3. All the subtrees of the left and right children observe the two rules.

Figure shows a binary data structure observing the above three rules. The number 7 is the root node of the binary tree. It has two sub-trees, the left subtree with node 3 and right subtree with node 9. The value of left subtree node is lower than the value of the root and the value of the right subtree node is higher than the value of the root.



## Searching and Inserting in Binary search Trees

Suppose T is a binary search tree, and suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the binary search tree T, or insert ITEM as a new node in its appropriate place in the tree.



- (a) Compare ITEM with the root node N of the tree

(i) if  $ITEM < N$ , proceed to the left child of  $N$ .

(ii) if  $ITEM > N$ , proceed to the right child of  $N$ .

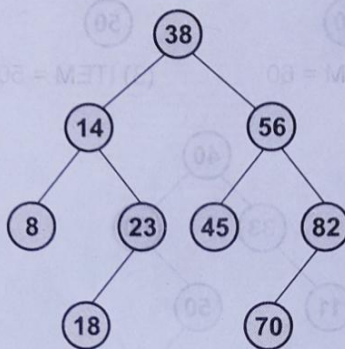
- (b) Repeat step (a) until one of the following occurs :

- (i) we meet a node  $N$  such that  $ITEM=N$ . In this case the search is successful.

- (ii) we meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

In other words, proceed from the root R down through the tree T until finding ITEM in T or inserting ITEM as a terminal node in T.

### Example : (Insertion)



Consider the binary tree T

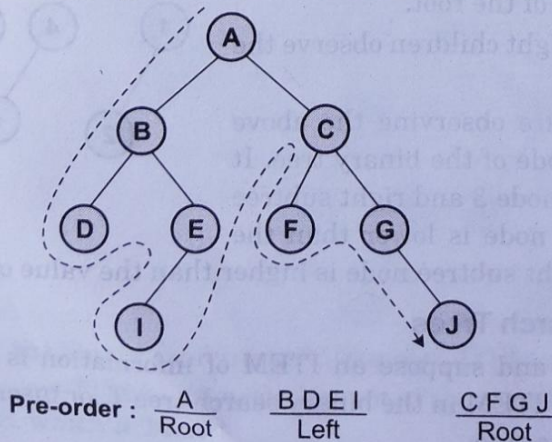
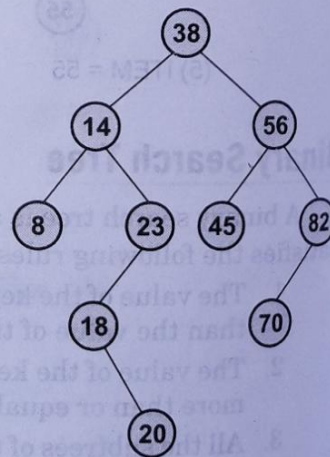
Suppose  $\text{ITEM} = 20$  is given. We obtain the following steps :

1. Compare ITEM = 20 with the root 38, of the tree T. Since  $20 < 38$ , proceed to the left child of 38, which is 14.

2. Compare ITEM = 20 with 14. Since  $20 > 14$ . Proceed to the right child of 14, which is 23.

3. Compare ITEM = 20 with 23. Since  $20 < 23$ , proceed to the left child of 23, which is 18.

4. Compare ITEM = 20 with 18. Since  $20 > 18$  and 18 does not have a right child, insert 20 as the right child of 18.



Pre-order :  $\frac{A}{\text{Root}}$        $\frac{BDEI}{\text{Left}}$        $\frac{CFGJ}{\text{Root}}$

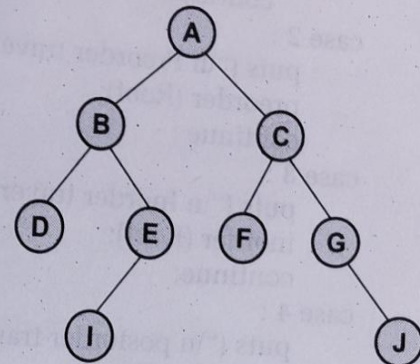


## post-order Traversal

The Post-order traversal of non-empty binary tree is defined as follows :

1. Traverse the left-subtree in Post order (Left).
2. Traverse the right subtree in Post order (Right).
3. Visit the Root node (Root).

That is, in a post-order traversal the left and right subtrees are recursively processed before visiting the root. The left subtree is taken up first and is traversed in Post-order. Then the right subtree is taken up and is traversed in Post-order. Finally, the data of the root node is displayed. The Post order traversal of the binary tree is shown in the fig.



Post-order :  $\frac{DIEB}{\text{Left}} \quad \frac{FJGC}{\text{Right}} \quad \frac{A}{\text{Root}}$

//Program of traversing to binary tree.

```

#include <stdio.h>
#include <conio.h>
struct node
{
    int num;
    struct node *left;
    struct node *right;
};
typedef struct _node node;
node* root=NULL;
node* insert (struct rec*tree, long num);
void preorder (node *tree);
void inorder (node* tree);
void postorder (node*tree);
int count =1;
void main ()
{
    int choice;
    long digit;
    do
    {
        choice=select ();
        switch (choice)
        {
            case 1 : puts ("Enter integer to put \n");
                    scanf ("%d",& digit);
                    while (digit!=0)
                    {
                        root=insert(root,digit);
                    }
                }
    }
  
```



```

        scanf("%d",& digit);
    }
    continue;
case 2 :
    puts ("\n Preorder traversing Tree");
    preorder (Root);
    continue ;
case 3 :
    puts ("\n Inorder traversing Tree");
    inorder (Root);
    continue;
case 4 :
    puts ("\n postorder traversing tree");
    postorder (Root);
    continue;
case 5:
    puts ("End");
    exit (0);
}
} which (choice!=5);
}
int select ()
{
    int selection;
    do
    {
        puts ("Enter 1 : Insert a node in the BT");
        puts ("Enter 2 : Display (Preorder) the BT");
        puts ("Enter 3 : Display (Inorder) the BT");
        puts ("Enter 4 : Display (Postorder) the BT");
        puts ("Enter 5 : END");
        puts ("Enter your choice");
        scanf("%d",& selection);
        if (( selection <1)!! (selection >5))
        {
            puts ("wrong choice : Try Again");
            getch ();
        }
        While ((selection>1)!! (selection<5));
        return (selection);
    }
    node * insert (node * P, long digit)
    {
        if (P== NULL)
        {
            P=(node *) malloc (size of (node));
            P→ left = P→ right =NULL;
            P→ num=digit ;
            count ++;

```



```

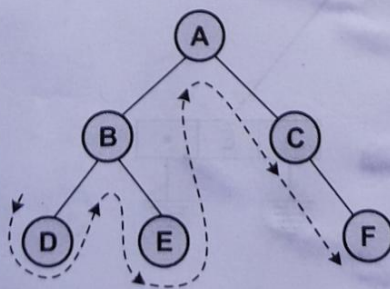
    }
    else
    if (count % 2 == 0)
        P→left = insert (P→left, digit);
    else
        P→right = insert (P→right, digit);
    return (P);
}

void preorder (node * P)
{
    if (P != NULL)
    {
        printf("%d\n", P→num);
        Preorder (P→left);
        Preorder (P→right);
    }
}

void inorder (node *P)
{
    if (P != NULL)
    {
        inorder (P→left);
        printf("%d\n", P→num);
        inorder (P→right);
    }
}

void Postorder (node *P)
{
    if (P != NULL)
    {
        preorder (P→left);
        preorder (P→right);
        printf("%d\n", P→num);
    }
}

```

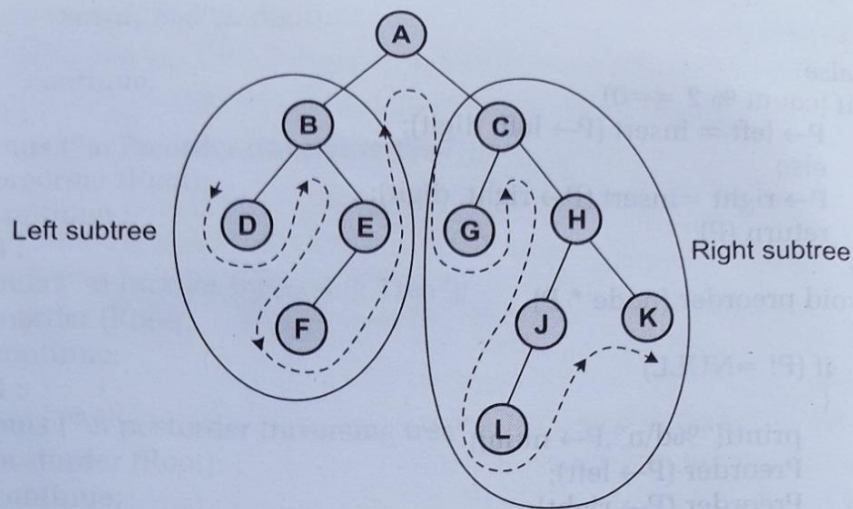


In-order :     $\frac{DBE}{\text{Left}}$      $\frac{A}{\text{Root}}$      $\frac{CF}{\text{Right}}$

**Another Example :**

**In-order**





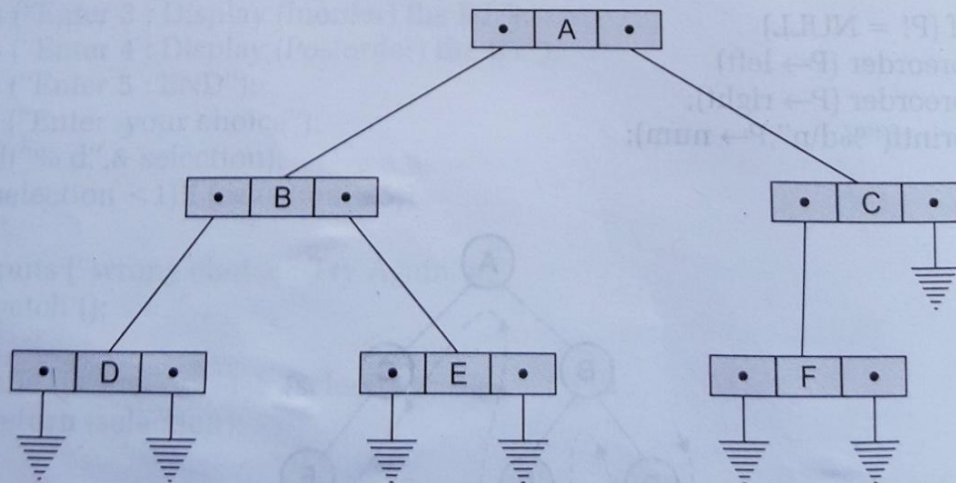
In-order :  $\frac{D B F E}{\text{Left}} \quad \frac{A}{\text{Root}} \quad \frac{G C L J H K}{\text{Right}}$

### Pre-order Traversal

The Preorder traversal of a non-empty binary tree is defined as :

1. Visit the root node (Root)
2. Traverse the left subtree in Pre-order (Left)
3. Traverse the right subtree in Pre-order (Right)

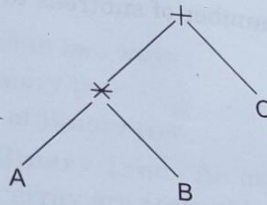
That is, in a Pre-order traversal the root node is visited before traversing its left and right subtrees. The Preorder notation is recursive in nature, so even within the left subtree and right subtree the above three steps are followed and its linked list representation is



### Traversal of a Binary Tree

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are many applications that essentially require traversal of binary trees. For example, a binary tree could be used to represent an arithmetic expression as shown below.





There are three standard ways of traversing a binary tree  $T$  with Root  $R$ . These algorithms are called Pre-order, In-order and Post-order.

### In-order Traversal

The in-order traversal of a non-empty binary tree is defined as follows :

1. Traverse the left subtree in inorder (Left)
2. Visit the root node (Root)
3. Traverse the right subtree in inorder (Right)

That is, in order traversal, the left subtree is traversed recursively in inorder before visiting the root node. After visiting the root node, the right subtree is taken up and it is traversed recursively again in inorder. The inorder traversal of the binary tree is shown in above figure.

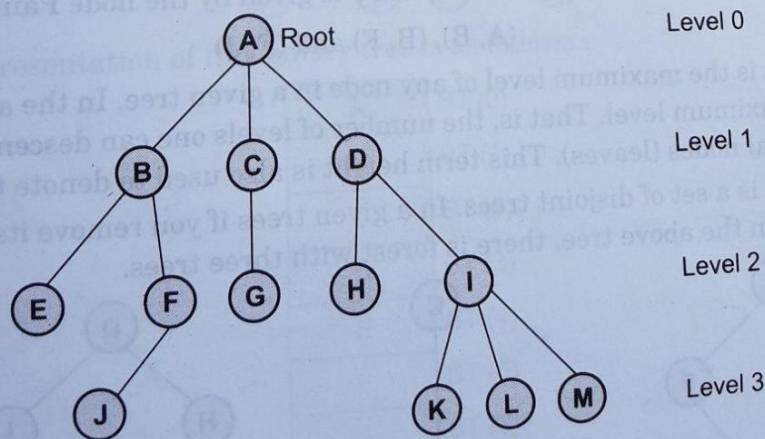
### Tree Terminologies

There are a number of terms associated with the trees which are listed below :

- |                  |                      |                     |                     |
|------------------|----------------------|---------------------|---------------------|
| 1. Root          | 2. Node              | 3. Degree of a Node | 4. Degree of a tree |
| 5. Terminal node | 6. Non-terminal node | 7. Siblings         | 8. Level            |
| 9. Edge          | 10. Path             | 11. Depth           | 12. Forest.         |

**1. Root :** It is specially designed data item in a tree. It is the first in the hierarchical arrangement of data items.

**Example :**



In the given tree, A is the Root item.

**2. Node :** Each data item in a tree is called a node. It is the basic structure in a tree. It specifies the data information and links (branches) to other data item. There are 13 nodes in the given tree.



**3. Degree of a node :** It is the number of subtrees to a node in a given tree. In the given tree,

- ❖ The degree of a node A is 3
- ❖ The degree of a node B is 2
- ❖ The degree of a node C is 1
- ❖ The degree of a node H is 0
- ❖ The degree of a node I is 3

**4. Degree of a tree :** It is the maximum degree in a given tree. In the given/above tree the node A has degree 3 and another node I is also having its degree 3. In all this value is the maximum. So the degree of the above tree is 3.

**5. Terminal node :** A node with degree zero (0) is called a terminal node or a leaf node. In the above tree, there are 7 (seven) terminal nodes. They are E, J, G, H, K, L and M.

**6. Non-terminal node :** Any node (except the root node) whose degree is not zero is called non-terminal node. Non-terminal nodes are the intermediate nodes in traversing the given tree from its root node to the terminal nodes (leaves). There are five (5) non-terminal nodes (B, C, D, F and I)

**7. Siblings :** The children nodes of a given Parent node are called Siblings. They are also called brothers. In the above tree,

E and F are siblings of Parent node B.

K, L and M are siblings of Parent node I,

**8. Level :** The entire tree structure is levelled in such a way that the root node is always at level 0. Then, its immediate children are at level 1 and their immediate children are at level 2 and so on up to the terminal nodes. In general, if a node is at level  $n$ , then its children will be at level  $n + 1$ . In the above tree, the level is 4.

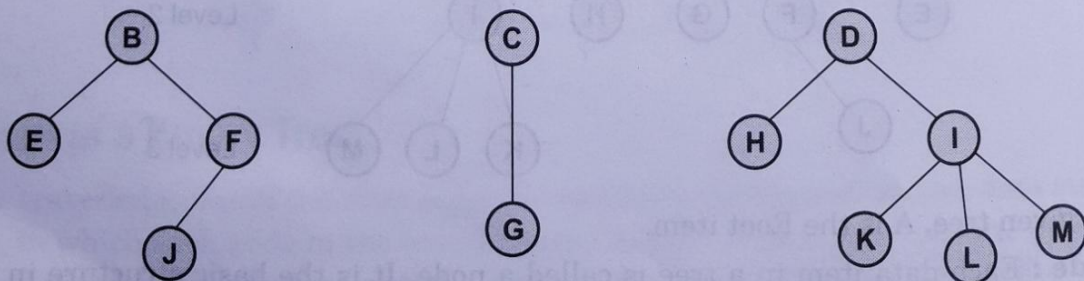
**9. Edge :** It is a connecting line of two nodes, that is, the line drawn from one node to another node is called an edge.

**10. Path :** It is a sequence of consecutive edges from the source node to the destination node. In the above tree, the Path between A and J is given by the node Pairs.

(A, B), (B, F) and (F, J)

**11. Depth :** It is the maximum level of any node in a given tree. In the above tree, the root node A has the maximum level. That is, the number of levels one can descend the tree from its root to the terminal nodes (leaves). This term height is also used to denote the depth.

**12. Forest :** It is a set of disjoint trees. In a given trees if you remove its root node, then it becomes a forest. In the above tree, there is forest with three trees.



Forest



## Binary Tree Representation

A Binary tree can be represented in two ways :

1. Array Representation of Binary tree
2. Linked list Representation of Binary tree

**1. Array Representation of Binary Tree :** An array can be used to store the nodes of binary tree. The nodes stored in an array are accessible sequentially. In C, Array starts with index 0 to MAXSIZE - 1. Here, numbering of binary tree node start from 0 rather than 1. The maximum number of nodes is specified by MAXSIZE.

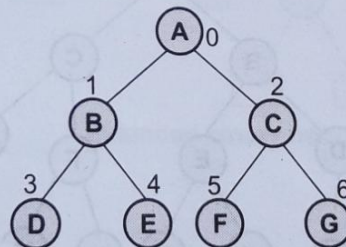
The root node is always at index 0. Then, in successive memory location the left child and right child are stored. Consider a binary tree with only 3 nodes as shown. Let BT denote a binary tree.

The Array representation of this binary tree is as follows :

|        |   |
|--------|---|
| BT (0) | A |
| BT (1) | B |
| BT (2) | C |

Binary tree

Here, A is the father of B and C. B is the left child of A and C is the right child of A. Let us extend the above tree by one more level as shown below.



The Array representation of this binary tree is as follows :

|   |        |
|---|--------|
| A | BT (0) |
| B | BT (1) |
| C | BT (2) |
| D | BT (3) |
| E | BT (4) |
| F | BT (5) |
| G | BT (6) |

## Linked list Representation

Binary tree also can be represented by using linked list. In this way, the basic component to be represented in a binary tree is a node. The node consists of three fields such as :

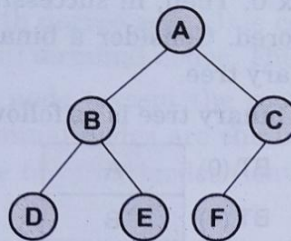


- ❖ Data
- ❖ Left child
- ❖ Right child

The data field holds the value to be given. The left child is a link field which contains the address of its left node and the right child contains the address of the right node.

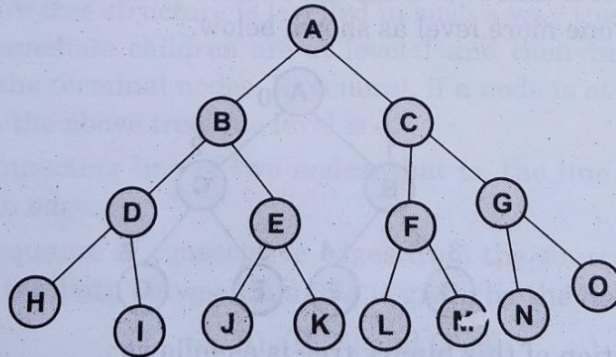
| Left child | Data | Right child |
|------------|------|-------------|
|------------|------|-------------|

Consider the following binary tree :



### Complete binary tree

A binary tree is said to be a complete binary tree if all the leaf nodes of the tree are at same level. Thus, the tree has maximum number of nodes at all the levels. For example,



at any level  $n$  of binary tree, there can be at the most  $2^n$  nodes, that is :

At  $n = 0$ , there can be at most  $2^0 = 1$  node

$n = 1$ , there can be at most  $2^1 = 2$  nodes

$n = 2$ , there can be at most  $2^2 = 4$  nodes

: : : :  
: : : :

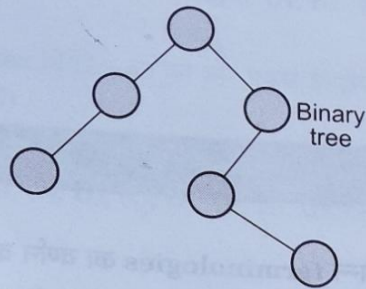
At level  $n$ , there can be at most  $2^n$  nodes.

### Extended Binary tree

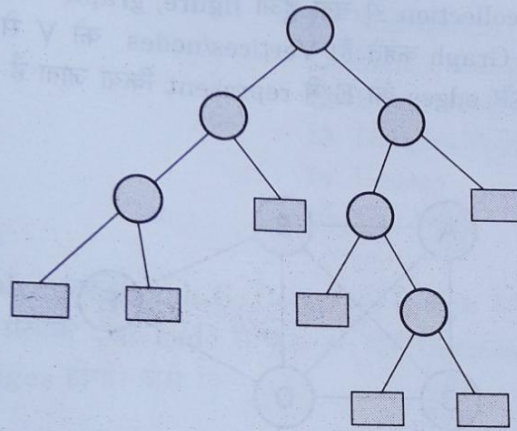
A binary tree  $T$  is said to be a 2-tree or an extended binary tree if each node  $N$  has either 0 or 2 children. In such a case, the nodes with 2 children are called internal nodes and the nodes with 0 children are called external node. Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes.



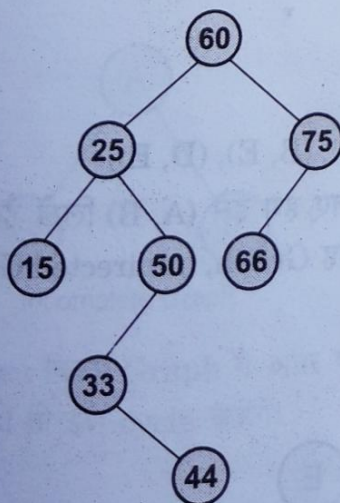
The term “Extended binary tree” comes from the following operation. Consider any binary tree  $T$ , such as the tree,



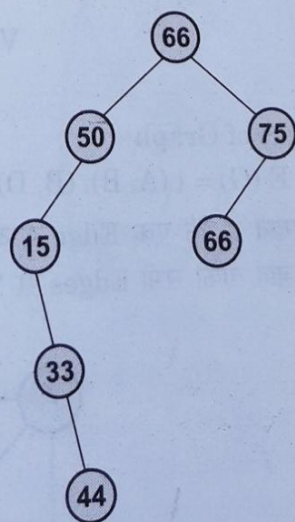
Then  $T$  may be “converted” into a 2-tree by replacing each empty subtree by a new node.



**Case 3 :**  $N$  has two children.



Before deletion



After deletion





## Unit

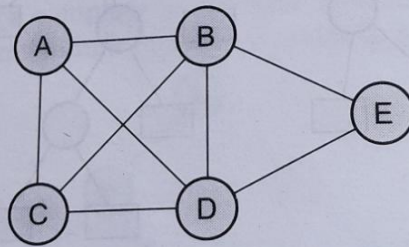
## 7

## Graph

प्रश्न 1. Graph क्या है? इसके विभिन्न terminologies का वर्णन कीजिए।

उत्तर—**Graph** : Graph एक non-linear Data structure है। या यह कहें कि nodes या vertices का collection और एक edges के collection से बना हुआ figure, graph कहलाता है। या यह कहें कि set of vertices और set of edges को Graph कहते हैं। Vertices/nodes को V से दर्शाते हैं जो Data को store करने के लिए use किया जाता है और edges को E से represent किया जाता है जो किन्हीं दो vertex को जोड़ने के लिए use किया जाता है।

**Example :**



इस चित्र के द्वारा एक undirected Graph को दर्शाया (Represent) गया है जिसमें vertices या nodes निम्न हैं A, B, C, D और E.

या

$$V(G) = (A, B, C, D, E)$$

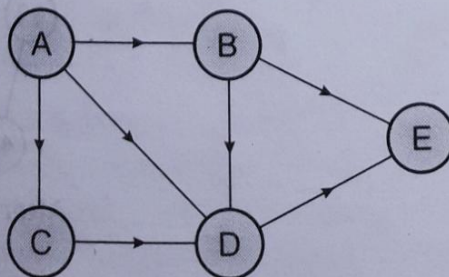
और

**Edges of Graph**

$$E(G) = \{ (A, B), (B, D), (D, C), (C, A), (A, D), (B, E), (D, E) \}$$

यहाँ पर हम देखते हैं कि एक Edge A और B को जोड़ती है इसलिए हम इसे (A, B) लिखे हैं। इसे (B, A) भी लिख सकते हैं। यही बात बाकी सभी Edges पर भी लागू होती है। इसीलिए यह Graph, undirected Graph कहलायेगा।

**Example :**





यह एक Directed Graph है जहाँ पर हर node दूसरे node से Directed edge से जुड़ा होता है यदि हम  $E = (A, B)$  लेते हैं तो Edge A से प्रारम्भ होगा और B पर समाप्त होगा। यहाँ पर A को Tail या initial Vertex तथा B को Head या Final node/vertex कहा जाता है। अतः  $(A, B)$  और  $(B, A)$  दो Different Edges को represent करेंगे।

यहाँ पर दिये गये Graph में Vertex of Graph को इस प्रकार लिखेंगे।

$$V(G) = (A, B, C, D, E)$$

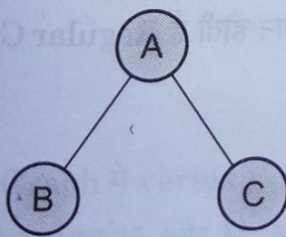
और Edges Graph को इस प्रकार लिखेंगे—

$$E(G) = \{(A, B), (A, C), (A, D), (C, D), (B, E), (B, D), (D, E)\}$$

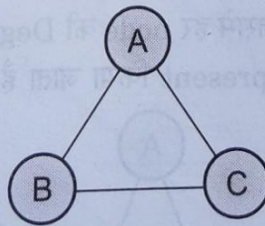
## Terminology of Graph

1. Complete Graph
2. Cycle
3. Labelled Graph
4. Weighted Graph
5. Regular Graph
6. Parallel Edge
7. Self loop
8. Adjacent Vertices
9. Path
10. Multigraph
11. Source and Sink
12. Direct Acyclic Graph
13. Isolated Vertex
14. Degree
15. Incidency

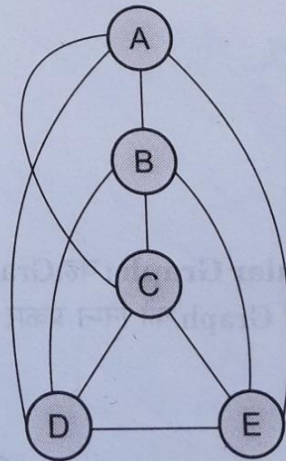
**1. Complete Graph :** एक graph G, Complete Graph कहलायेगा, यदि Graph की प्रत्येक Node, Graph का प्रत्येक दूसरे node से जुड़ा हो। एक Complete Graph जिसमें  $n$  vertices हैं उसमें  $n(n-1)/2$  Edges होगी। जैसे कि—



Incomplete Graph



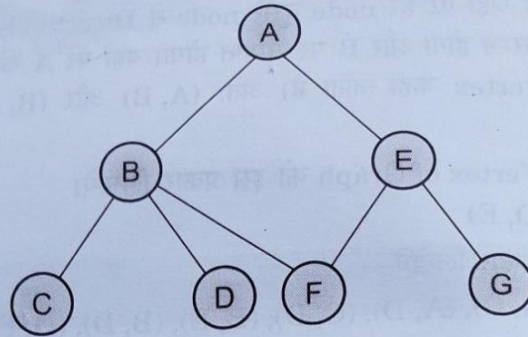
Complete Graph



Complete Graph

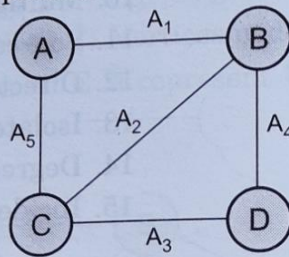
**2. Cycle :** किसी Graph में अगर एक vertex से Traversing शुरू करके उसी vertex पर traversing खत्म हो तो इसे cycle कहेंगे।



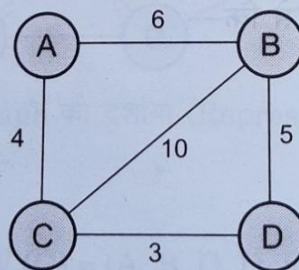


यहाँ पर Path A से शुरू होकर A पर ही खम होगा  $(A \rightarrow B)(B \rightarrow F)(F \rightarrow E)(E \rightarrow A)$

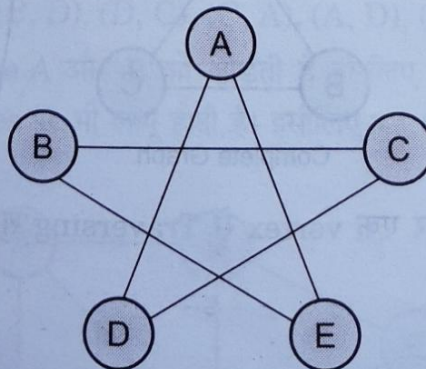
3. **Labelled Graph :** एक Graph जिसकी Edge पर Data लिखा होता है labelled Graph कहलाता है। ऐसे Graph को निम्न प्रकार से represent किया जाता है—



4. **Weighted Graph :** एक Graph जिसकी Edges पर धनात्मक अंक लिखे होते हैं उन्हें weighted Graph कहते हैं। ऐसे Graph को निम्न प्रकार से represent किया जाता है—



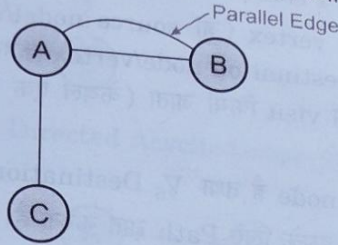
5. **Regular Graph :** वह Graph, जिसमें हर node की Degree समान होती है Regular Graph कहलाता है। ऐसे Graph को निम्न प्रकार से represent किया जाता है—



यहाँ A, B, C, D, E सभी vertex की Degree 2 है।

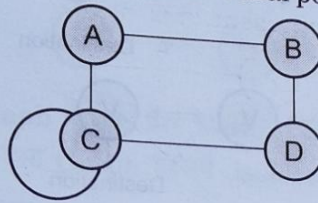


- 6. Parallel Edges :** ऐसे Edges जिनका Initial point और End point समान होता है उन्हें parallel edges कहते हैं। ऐसे edge को निम्न प्रकार से represent किया जाता है—



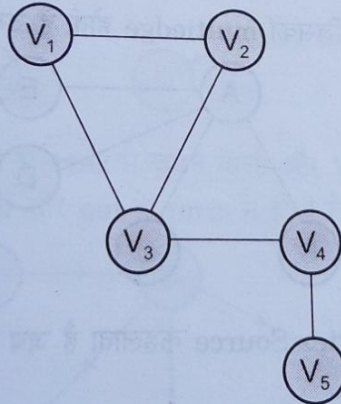
यहाँ पर A और B Vertex को एक साथ दो समान Edge जोड़ रहे हैं उन Edges को Parallel Edges कहेंगे।

- 7. Self loop :** ऐसा node जिसका Initial point और Final point एक ही होता है उसे Self loop कहते हैं।



यहाँ पर एक edge C Vertex से शुरू होकर C पर ही खत्म हो रहा है इसलिए यह edge, Self Loop कहलायेगा।

- 8. Adjacent Vertices (Neighbours) :** किसी एक vertex से जुड़ने वाली दूसरी vertex पहली vertex का Adjacent कहलायेगा। जैसे कि



इस Graph में vertex  $V_1$ , Vertex  $V_2$  के Adjacent कहलायेगा, यदि एक edge  $(V_1, V_2)$  या  $(V_2, V_1)$  है तो यहाँ पर  $V_1$  और  $V_2$  Adjacent हैं,  $V_1$  और  $V_3$  adjacent हैं,  $V_2$  और  $V_3$  adjacent हैं,  $V_3$  और  $V_4$  adjacent हैं तथा  $V_4$  और  $V_5$  adjacent हैं क्योंकि ये सभी Vertices किसी न किसी Edge के द्वारा आपस में जुड़े हुए हैं। प्रत्येक Vertex को उससे सम्बद्ध edges के आधार पर Degree प्रदान की जाती है। यहाँ पर  $V_1$  Vertex से, दो Vertex  $V_2$  और  $V_3$  जुड़े हैं इसलिए  $V_1$  की Degree 2 होगी। इसी प्रकार,

Vertex  $V_2$  की Degree = 2

Vertex  $V_3$  की Degree = 3

Vertex  $V_4$  की Degree = 2

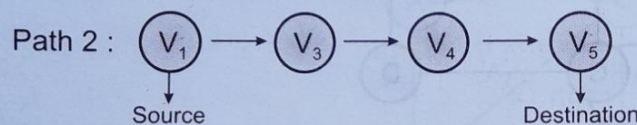
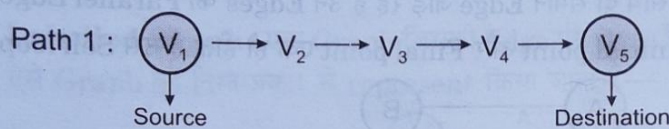


Vertex  $V_5$  की Degree = 1

वह Vertex जिसकी Degree केवल 1 होती है उसे Pandent vertex कहा जाता है।

**9. Path :** किसी Graph में एक vertex (जो source node/vertex कहलाएगा) से दूसरे vertex (जो कि Destination node/vertex कहलायेगा) तक जाने के लिए जिन nodes/vertex को visit किया जाता (केवल एक बार) है वह path कहलाता है। जैसे कि—

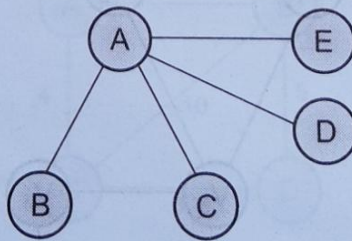
यहाँ मान लिया जाए कि  $V_1$  source node है तथा  $V_5$  Destination node है (अर्थात् के  $V_1$  से  $V_5$  तक जाना है) तो इसके लिए Path ज्ञात करना है। यहाँ पर दो Path हो सकते हैं।



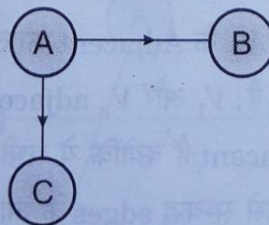
इन दोनों Paths में से जो Path सबसे छोटा होता है इसे अच्छा Path माना जाता है। यहाँ पर Path 2, Path 1 से छोटा है।

Path 2  $\rightarrow (V_1, V_3), (V_3, V_4), (V_4, V_5)$

**10. Multigraph :** एक Graph जिसकी multiedge होती है उस graph को multigraph कहा जाता है।

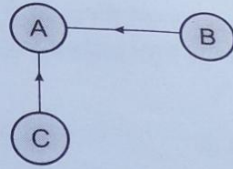


**11. Source and Sink :** एक Vertex Source कहलाता है जब उसकी Out-degree 0 (शून्य) से ज्यादा हो तथा In-degree=0 हो। जैसे कि—

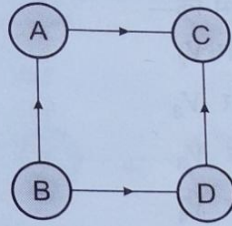


एक vertex, Sink कहलाता है जब इस vertex की In-degree 0 (शून्य) से ज्यादा हो तथा Out-degree=0 हो। जैसे कि—

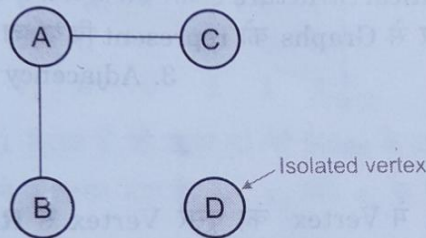




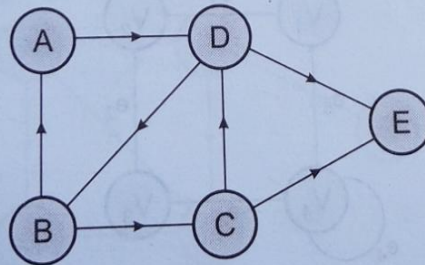
**12. Direct Acyclic Graph :** Directed Acyclic Graph एक Directed Graph है जिसमें Cycles नहीं बनती है।



**13. Isolated Vertex :** किसी Graph में एक ऐसा Vertex जो किसी अन्य Vertex के Contact में नहीं होता, वह isolated Vertex कहलाता है। ऐसा Graph जिसमें isolated Vertex होता है, उसे NULL Graph या Disconnected Graph कहते हैं; जैसे कि—



**14. Degree :** एक Graph में, किसी Vertex से जुड़ने वाली और Vertex उस Vertex की Degree कहलाती है। परन्तु Directed Graph में Degree in और out के form में होती है। जैसे कि—



**In-degree :** Vertex की ओर Direction वाली Edges को In-degree कहते हैं जैसे कि दिये गये Graph में

In-degree A = 1

In-degree B = 1

In-degree C = 1

In-degree D = 2

In-degree E = 2



### Out Degree होगी

Out-degree A = 1

Out-degree B = 2

Out-degree C = 2

Out-degree D = 2

Out-degree E = 0 होगी।

**15. Incidency :** एक Edge जिन vertices को जोड़ती है वे vertices उस edge की incident होती हैं। जैसे कि—

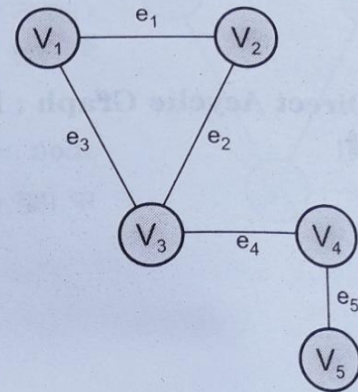
Edge  $e_1$  की Incident = vertex  $V_1$  और  $V_2$

Edge  $e_2$  की Incident = vertex  $V_2$  और  $V_3$

Edge  $e_3$  की Incident = vertex  $V_1$  और  $V_3$

Edge  $e_4$  की Incident = vertex  $V_3$  और  $V_4$

Edge  $e_5$  की Incident = vertex  $V_4$  और  $V_5$



**प्रश्न 2. Graph को memory में कितने प्रकार से represent कर सकते हैं?**

उत्तर—Graph एक Mathematical Structure है और इसका प्रयोग कई सारे Problems को solve करने के लिए किया जाता है। यहाँ पर तीन प्रकार से Graphs को represent किया जा सकता है।

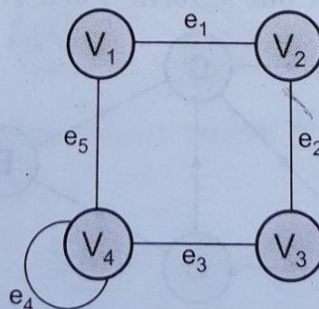
1. Adjacency matrix

3. Adjacency list Representation

2. Incidency matrix

### 1. Adjacency Matrix

इस प्रकार के Representation में Vertex का दूसरे Vertex से Relation को एक Matrix के द्वारा represent करते हैं। दिये गये Graph का adjacency matrix representation को निम्न प्रकार से दर्शाया जा सकता है—



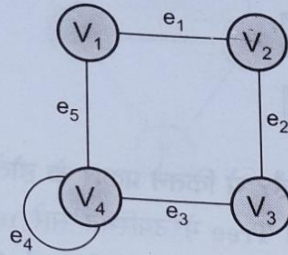
|       | $V_1$ | $V_2$ | $V_3$ | $V_4$ |
|-------|-------|-------|-------|-------|
| $V_1$ | 0     | 1     | 0     | 1     |
| $V_2$ | 1     | 0     | 1     | 0     |
| $V_3$ | 0     | 1     | 0     | 1     |
| $V_4$ | 1     | 0     | 1     | 1     |



यहाँ हमने उन Vertices के नीचे 1 लिखा है जो सामने किसी vertex की Adjacent है। शेष के नीचे 0 (शून्य) लिखा है। जैसे  $V_1$ ,  $V_2$  और  $V_4$  की Adjacent है इसलिए हमने  $V_2$  और  $V_4$  के नीचे 1 लिखा है और  $V_1$  और  $V_3$  के लिए 0 (शून्य) लिखा है।

## 2. Incidency Matrix

इस प्रकार के representation में Vertex का विभिन्न Edges से Relation को एक matrix के द्वारा प्रस्तुत करते हैं। जैसे कि—



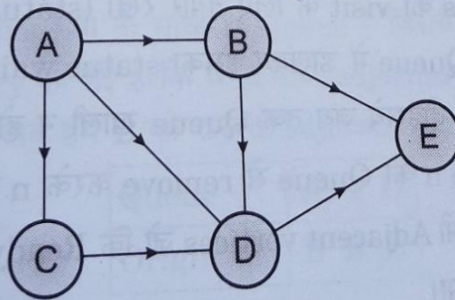
इस Graph के लिए Incidency matrix निम्न प्रकार से represent कर सकते हैं—

|       | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ |
|-------|-------|-------|-------|-------|-------|
| $V_1$ | 1     | 0     | 0     | 0     | 1     |
| $V_2$ | 1     | 1     | 0     | 0     | 0     |
| $V_3$ | 0     | 1     | 1     | 0     | 0     |
| $V_4$ | 0     | 0     | 1     | 1     | 1     |

यहाँ हमने उन vertices के आगे 1 लिखा है जो ऊपर दी गई Edge के incident है। जैसे  $V_1$ ,  $e_1$  और  $e_5$  की incident है इसलिए  $e_1$  और  $e_5$  के नीचे 1 लिखा गया है।  $e_2$ ,  $e_3$  और  $e_4$  के नीचे 0 (शून्य) लिखा गया है।

## 3. Adjacency List Representation

Graph में Adjacency List representation को represent करने के लिए सबसे पहले एक table बनायेंगे। जिसमें प्रत्येक node की adjacent node, उसके सामने लिखी हों। जैसे कि एक Graph लेते हैं—

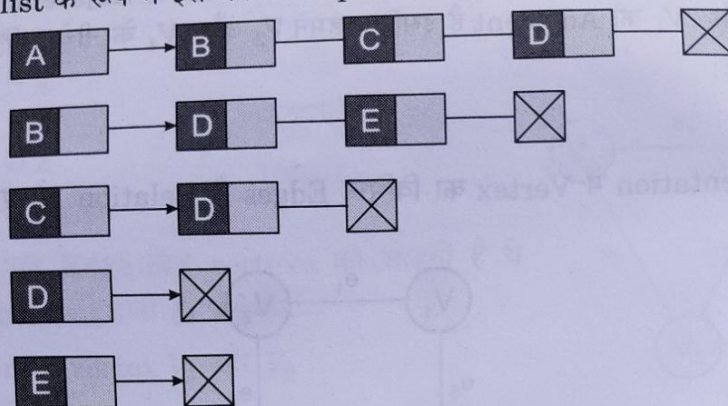


सबसे पहले Adjacency table बनायेंगे।

| A | B | C | D |
|---|---|---|---|
| B | D | E |   |
| C | D |   |   |
| D | — |   |   |
| E | — |   |   |



अब इस Table को list के रूप में इस प्रकार Represent करेंगे—



**प्रश्न 3. Graph traversal क्या होता है? ये कितने प्रकार के होते हैं?**

**उत्तर—**Traversal means Graph या Tree में उपस्थित सारे nodes को कम से कम एक बार जरूर visit किया जाना। Graph के हर vertex को एक बार visit करना Graph traversal कहलाता है। Graph traversal के लिए दो प्रकार के तरीके का प्रयोग किया जाता है—

1. Breadth First Search (B F S)
2. Depth First Search (D F S)

**1. Breath First Search (B F S) :** जैसा कि नाम से ही पता चल रहा है कि इस तरीके में nodes को चौड़ाई से Visit करना है। B F S में पहले हम start vertex की समस्त vertices को visit करते हैं और फिर इनकी सभी unvisited vertices को visit करते हैं।

इसी प्रकार आगे तक समस्त Vertices को visit किया जाता है। B F S के लिए निम्नलिखित Algorithm को ध्यान में रखते हैं—

Status 1 = Ready

Status 2 = Waiting

Status 3 = Process

Step 1 : Graph की सभी nodes को visit के लिए तैयार रखें। (status 1)

Step 2 : Start Vertex A को Queue में डालकर इसका status waiting करें। (status 2)

Step 3 : Step 4 से Step 5 तक दोहरायें जब तक Queue खाली न हो जाए।

Step 4 : Queue की पहली node n को Queue से remove करके n को process कर दें। (status 3)

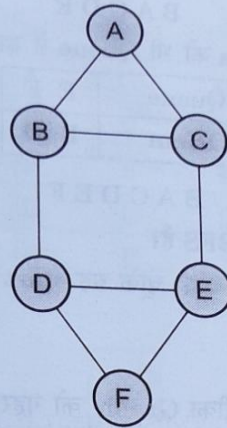
Step 5 : Queue के अन्त में n के सभी Adjacent vertices जो कि Ready state में हैं, को डाल दें। (status 1)

Step 6 : Step 3 का लूप खत्म हुआ।

Step 7 : Exit.

इस पूरे Process को समझने के लिए एक example लिया गया है।





इस Graph के लिए पहले Adjacent Table बनाते हैं।

| A | B | C |   |
|---|---|---|---|
| B | A | C | D |
| C | A | B | E |
| D | B | E | F |
| E | C | D | F |
| F | D | E |   |

अब हम किसी भी vertex को start vertex बनाकर कार्य शुरू कर सकते हैं। इस Example में हमने B को start vertex माना है। अब हमें इसे Queue में डालना है।

| Queue  | B |
|--------|---|
| Origin | O |

इस Queue के दो भाग हैं—पहला भाग तो साधारण Queue है जिसमें हम vertex की Adjacent Vertices डालेंगे, जबकि दूसरे भाग में यह entry की गई है कि वे किसकी adjacent vertices हैं। जो-जो vertices, visit हो जाए उन्हें Queue के नीचे लिख लें। अब हमें B की समस्त Adjacent vertices को Queue में डालना है।

| Queue  | A C D |
|--------|-------|
| Origin | B B B |

B A C D

अब इन Vertex की unvisited Adjacent vertices को visit करना है। जैसा कि हम देख सकते हैं कि A की समस्त Adjacent vertices visit हो चुकी हैं, इसलिए अब हम C की बची Adjacent vertices को Queue में डालेंगे।

| Queue  | D E |
|--------|-----|
| Origin | B C |



B A C D E

अब हम D की बची Adjacent Vertices को भी Queue में डालना है।

|        |     |
|--------|-----|
| Queue  | E F |
| Origin | B D |

B A C D E F

B A C D E F यह दिये गये Graph का BFS है।

नोट—एक Graph का BFS भिन्न हो सकता है चूँकि यह start vertex पर निर्भर करता है।

## Depth First Search (DFS)

इसके नाम से ही पता लग रहा है कि यह तरीका Graph को गहराई से visit करता है। DFS में हम सबसे पहले start vertex को stack के द्वारा visit करते हैं। फिर इसकी समस्त Adjacent vertices को stack में डालकर stack top पर स्थित क्रिया तब तक दोहराते हैं जब तक कि stack खाली न हो जाए। DFS करने के लिए निम्न Algorithm को करते हैं।

**Step 1 :** Graph की सभी nodes को visit के लिए तैयार रखें। (status 1)

**Step 2 :** Start vertex को stack में डालकर इसका status waiting कर दें। (status 2)

**Step 3 :** Step 3 से 5 तक तब तक दोहरायें जब तक कि stack खाली न हो जाए।

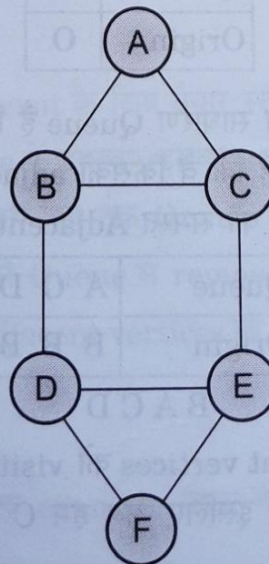
**Step 4 :** Stack के top में से node को निकालकर इसे process करें। (Status 3)

**Step 5 :** n की Adjacent vertices को stack में डालकर इसका status 1 से 2 करें।

**Step 6 :** Step 3 का loop खत्म हुआ।

**Step 7 :** Exit.

इस पूरे Process को समझने के लिए निम्न example को देखते हैं—

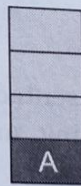


सबसे पहले Adjacent table बनाते हैं।



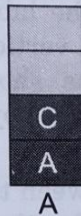
| A | B | C |   |
|---|---|---|---|
| B | A | C | D |
| C | A | B | E |
| D | B | E | F |
| E | D | C | F |
| F | D | E |   |

अब start vertex से stack में डाल दें।

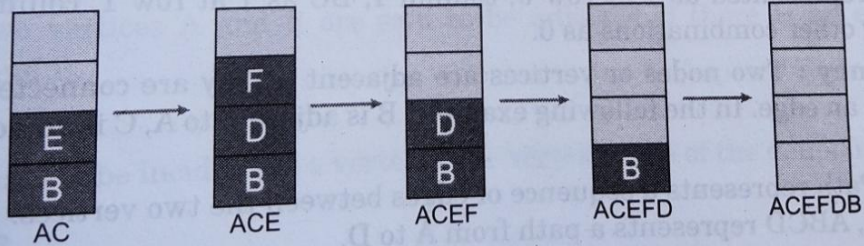


अब सबसे पहले visited node को stack के नीचे लिख लें।

Visited node की समस्त Adjacent nodes को stack में डाल दें।



अब पुनः Top of the stack को stack से बाहर निकालकर इसकी Adjacent को stack में डाल दें और इसी प्रकार तब तक आगे करते रहिए जब तक stack खाली ना हो जाए।



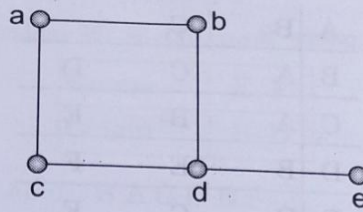
अब Graph DFS में है : A, C, E, F, D, B

## Graph

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets  $(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, connecting the pairs of vertices. Take a look at the following graph :





In the above graph,

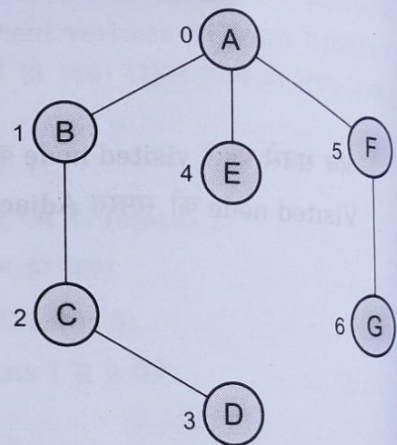
$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

## Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms :

- ❖ **Vertex** : Each node of the graph is represented as a vertex. In the following example, the labelled circle represent vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0, B can be identified using index 1 and so on.
- ❖ **Edge** : Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represent edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- ❖ **Adjacency** : Two nodes or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- ❖ **Path** : Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



## Graph Terminology

We use the following terms in graph data structure :

### Vertex

An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

### Edge

An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex). For example, in above graph, the link



between vertices A and B is represented as (A, B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B, D), (B, E), (C,D), (D,E)).

### Edges are three types :

1. **Undirected Edge** : An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B, then edge (A, B) is equal to edge (B, A).
2. **Directed Edge** : A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).
3. **Weighted Edge** : A weighted edge is an edge with cost on it.

### Undirected Graph

A graph with only undirected edges is said to be undirected graph.

### Directed Graph

A graph with only directed edges is said to be directed graph.

### Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

### End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

### Origin

If an edge is directed, its first endpoint is said to be origin of it.

### Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

### Adjacent

If there is an edge between vertices A and B, then both A and B are said to be adjacent. In other words, two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

### Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

### Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

### Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

### Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

### Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.



**Outdegree**

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

**Parallel edges or Multiple edges**

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

**Self-loop**

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

**Simple Graph**

A graph is said to be simple if there are no parallel and self-loop edges.

**Path**

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

**Graph Representations**

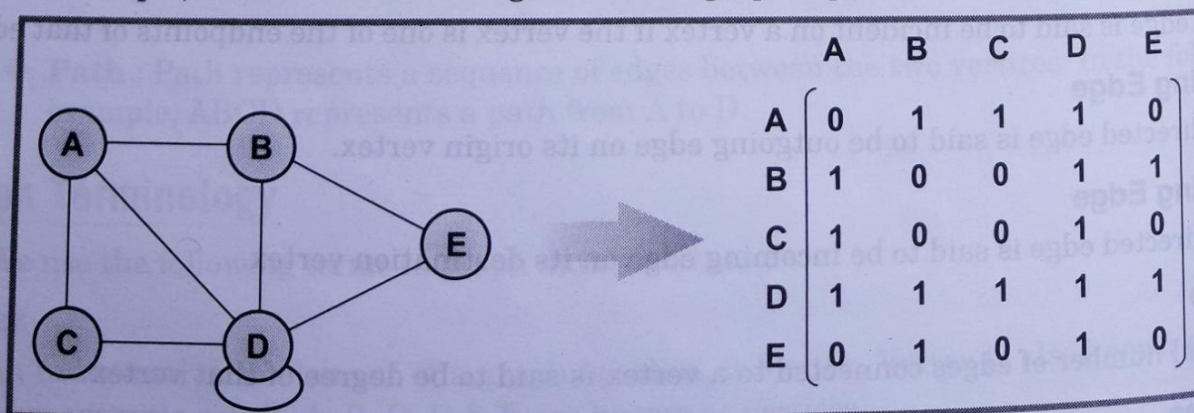
Graph data structure is represented using following representations :

1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

**Adjacency Matrix**

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of  $4 \times 4$  class. In this matrix, rows and columns both represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation :



Directed graph representation :



## Advantages and disadvantages of graph

### Advantages :

1. Allows easier processing of data.
2. It allows information stored on disk very efficiently.
3. These are necessary for designing an efficient algorithm.
4. It provides management of databases like indexing with the help of hash tables and arrays.
5. We can access data anytime and anywhere.
6. It is secure way of storage of data.
7. Graphs model real life problems.
8. It allows processing of data on software system.

### Disadvantages :

1. It is applicable only for advanced users.
2. If any issue occurs it can be solved only by experts.
3. Slow access in case of some data types.



# Experiments/Practicals

**// WAP to insert elements in an array.**

```
#include<stdio.h>
#include<conio.h>
#define MAXSIZE 50
void main()
{
    int x[MAXSIZE],i,n;
    clrscr();
    printf("enter size of array");
    scanf("%d",&n);
    printf("\n enter %d elements in the array:",n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&x[i]);
    }
    printf("\n elements in the array are:");
    for(i=0;i<n;i++)
    {
        printf("%d",x[i]);
    }
    getch();
}
```

**//WAP to delete an element from the specified index in the array.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int y[100],position,c,n;
    clrscr();
    printf("enter the numbers of elements in array ");
```



```

scanf("%d",&n);
printf("\n enter %d elements \n",n);
for(c=0;c<n;c++)
{
scanf("%d",&y[c]);
}
printf("enter the location where you wish to delete elements\n");
scanf("%d",&position);
if(position>=n+1)
{
printf("deletion is not possible\n");
}
else
{
for(c=position-1;c<n-1;c++)
{
y[c]=y[c+1];
}
}
printf("\n resultant array is \n");
for(c=0;c<n-1;c++)
{
printf("%d\n",y[c]);
}
getch();
}

```

#### **//WAP to perform push, pop and display operations on a stack.**

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 5
int top=-1,stack[MAX];
void push();
void pop();
void display();
void main()
{
int ch;
while(1)
{
printf("\n stack menu \n");
printf("\n\n1.push\n2.pop\n3.displa\n4.exit");
printf("\n enter your choice(1-4)");

```



```

scanf("%d",&ch);
switch(ch)
{
case 1:
push();
break;
case 2:
pop();
break;
case 3:
display();
break;
case 4:
exit(0);
break;
default:
printf("\wrong choice");
}
}
}
void push()
{
int val;
if(top==MAX-1)
{
printf("\n stack is full");
}
else
{
printf("\n enter element to push");
scanf("%d",&val);
top=top+1;
stack[top]=val;
}
}
void pop()
{
if(top== -1)
{
printf("\n stack is empty");
}
else
{
printf("\n deleted element is %d ",stack[top]) ;
}
}

```



```

top=top-1;
}
}
void display()
{
int i;
if(top== -1)
{
printf("\n stack is empty");
}
else
{
printf("\n stack is...\n");
for(i=top; i>=0; i--)
{
printf("%d\n", stack[i]);
}
}
}

```

**//WAP to sort the given elements in descending order by using insertion sort.**

```

#include<stdio.h>
#include<conio.h>
void main()
{
int data[100], n, temp, i, j;
printf("enter number of items");
scanf("%d", &n);
printf("enter elements");
for(i=0; i<n; i++)
{
scanf("%d", &data[i]);
}
for(i=1; i<n; i++)
{
temp=data[i];
j=i-1;
while(temp>data[j] && j>=0)
{
data[j+1]=data[j];
j--;
}
data[j+1]=temp;
}
printf("\n in ascending order");

```



```

for(i=0;i<n;i++)
{
printf("%d",data[i]);
}
getch();
}

```

### //WAP to sort the given elements by using bubble sort.

```

#include<stdio.h>
#include<conio.h>
void main()
{
int a[50],n,i,j,temp;
printf("enter the size of array:");
scanf("%d",&n);
printf("enter the array elements:");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
for(i=1;i<n;i++)
{
for(j=0;j<(n-1);j++)
{
if (a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
printf("\n array after sorting");
for(i=0;i<n;i++)
{
printf("%d",a[i]);
}
}

```

### //WAP to sort the given array by using merge sort.

```

#include<stdio.h>
#include<conio.h>
#define max 10
int a[11]={33,55,11,99,66,22,23,15,63,86,22};

```



```

int b[10];
void merging(int low, int mid, int high)
{
    int l1, l2, i;
    for(l1=low, l2=mid+1, i=low; l1<=mid && l2<=high; i++)
    {
        if(a[l1]<=a[l2])
            b[i]=a[l1++];
        else
        {
            b[i]=a[l2++];
        }
    }
    while(l1<=mid)
        b[i++]=a[l1++];
    while(l2<=high)
        b[i++]=a[l2++];
    for(i=low; i<=high; i++)
        a[i]=b[i];
}

void sort(int low, int high)
{
    int mid;
    if (low < high)
    {
        mid=(low+high)/2;
        sort(low, mid);
        sort(mid+1, high);
        merging(low, mid, high);
    }
    else
    {
        return;
    }
}

int main()
{
    int i;
    printf("list before sorting\n");
    for(i=0; i<=max; i++)
    {
        printf("%d", a[i]);
        sort(0, max);
    }
}

```



```

printf("\n list after sorting\n");
for(i=0;i<=max;i++)
{
printf("%6d",a[i]);
}
getch();
}

```

### //WAP to sort the given elements by using heap sort.

```

#include<stdio.h>
#include<conio.h>
void main()
{
int heap[10],no,i,j,c,root,temp;
printf("\n enter no of elements:\n");
scanf("%d",&no);
printf("\n enter numbers:\n");
for(i=0;i<no;i++)
{
scanf("%d",&heap[i]);
}
for(i=1;i<no;i++)
{
c=i;
do
{
root=(c-1)/2;
if(heap[root]<heap[c])
{
temp=heap[root];
heap[root]=heap[c];
heap[c]=temp;
}
c=root;
}
while(c!=0);
}
printf("heap array:\n");
for(i=0;i<no;i++)
{
printf("%d\n",heap[i]);
}
for(j=no-1;j>=0;j--)
{

```

```

temp=heap[0];
heap[0]=heap[j];
heap[j]=temp;
root=0;
do
{
c=2*root+1;
if((heap[c]<heap[c+1])&&c<j-1)
c++;
if(heap[root]<heap[c]&&c<j)
{
temp=heap[root];
heap[root]=heap[c];
heap[c]=temp;
}
root=c;
}
while(c<j);
}
printf("\n the sorted array is:\n");
for(i=0;i<no;i++)
{
printf("%d\n",heap[i]);
}
getch();
}

```

### //WAP to search an item by using binary search.

```

#include<stdio.h>
#include<conio.h>
void binary_search();
int a[50],n,item,loc,beg,mid,end,i;
void main()
{
printf("\n enter size of an array");
scanf("%d",&n);
printf("\n enter elements of the array in sorted form\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("\n enter item to be searched");
scanf("%d",&item);

```



```

binary_search()
{
    getch();
    void binary_search()
    {
        beg=0;
        end=n-1;
        mid=(beg+end)/2;
        while((beg<=end)&& (a[mid]!=item))
        {
            if(item<a[mid])
                end=mid-1;
            else
                beg=mid+1;
            mid=(beg+end)/2;
            printf("\n item found at location %d",mid+1);
        }
        if(a[mid]==item)
            printf("\n item does not exist");
    }
}

```

#### //MAP to search the element by using linear search.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int a[20],i,x,n;
    printf("how many elements\n");
    scanf("%d",&n);
    printf("enter arrays elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\n enter elements to be seach\n");
    scanf("%d",&x);
    for(i=0;i<n;i++)
    {
        if(a[i]==x)
        {
            break;
        }
        if(i<n)
            printf("elements found at index %d",i);
        else
            printf("element not found");
    }
}

```

```

    getch();
}

```

#### //MAP to demonstrate the basic operations of queue.

```

#include<stdio.h>
#include<conio.h>
int queue[5];
int front, rear;
void initqueue();
void display();
void main()
{
    int choice, info;
    clrscr();
    initqueue();
    while(1)
    {
        clrscr();
        printf("\n*****menu*****\n");
        printf("\n1.insert an element in queue\n");
        printf("\n2.delete an element from queue\n");
        printf("\n3.display the element from queue\n");
        printf("\n4.exit\n");
        printf("enter your choice");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                if(rear<4)
                {
                    printf("enter the number");
                    scanf("%d",&info);
                    if(front==-1)
                    {
                        front=0;
                        rear=0;
                    }
                    else
                        rear=rear+1;
                    queue[rear]=info;
                }
                else
                    printf("queue is full");
            }
        }
    }
}

```



```

    getch();
    break;
case 2:
{
    int info;
    if(front==1)
    {
        info=queue[front];
        if (front==rear)
        {
            front=-1;
            rear=-1;
        }
        else
        {
            front=front+1;
        }
        printf("no deleted is=%d",info);
    }
    else
    {
        printf("queue is empty");
        getch();
        break;
    }
case 3:
    display();
    getch();
    break;
case 4:
    exit();
    break;
default:
    printf("you entered wrong choice");
    getch();
    break;
}
}
}
}
void initqueue()
{
    front=rear=-1;
}
void display()
{
    int i;
    for(i=front;i<=rear;i++)

```

```

{
    printf("%d\n",queue[i]);
}
}
}

```

### /WAP to perform basic operations on singly link list.

```

#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<stdlib.h>
struct node
{
    int value;
    struct node *next;
};
void insert();
void display();
void delete();
typedef struct node DATA_NODE;
DATA_NODE
*head_node,*first_node,*temp_node=0,*prev_node,next_node,temp_count;
int data;
void main()
{
    int option=0;
    clrscr();
    printf("operations of singly linked list\n");
    while(option<5)
    {
        printf("\n options\n");
        printf("\n1.insert into linked list\n");
        printf("\n2.delete from linked list\n");
        printf("\n3.display linked list\n");
        printf("\n4.count linked list\n");
        printf("others:exit()\n");
        printf("enter your option\n");
        scanf("%d",&option);
        switch(option)
        {
            case 1:
                insert();

```



```

break;
case 2:
    delete();
break;
case 3:
    display();
break;
default:
    break;
}
}
}
void insert()
{
    printf("\n enter element for insert linked list\n");
    scanf("%d", &data);
    temp_node=(DATA_NODE*)malloc(sizeof (DATA_NODE));
    temp_node->value=data;
    if(first_node==0)
    {
        first_node=temp_node;
    }
    else
    {
        head_node->next=temp_node;
    }
    temp_node->next=0;
    head_node=temp_node;
    fflush(stdin);
}
void delete()
{
    int countvalue,pos,i=0;
    temp_node=first_node;
    printf("\n display linked list: \n");
    printf("\n enter position for delete element: \n");
    scanf("%d", &pos);
    if(pos>0 && pos <=countvalue)
    {
        if(pos==1)
        {
            temp_node=temp_node->next;
            first_node=temp_node;
            printf("\n deleted successfully\n");

```

```

    }
    else
    {
        while(temp_node!=0)
        {
            if(i==(pos-1))
            {
                prev_node->next=temp_node->next;
                if(i==(countvalue-1))
                {
                    head_node=prev_node;
                }
                printf("\n deleted successfully\n");
                break;
            }
            else
            {
                i++;
                prev_node=temp_node;
                temp_node=temp_node->next;
            }
        }
        else
        {
            printf("\n invalid position\n");
        }
        void display()
        {
            int count=0;
            temp_node=first_node;
            printf("\n display linked list\n");
            while(temp_node!=0)
            {
                printf("%d",temp_node->value);
                count++;
                temp_node=temp_node->next;
            }
            printf("\n no of items in linked list:%d\n",count);
        }
    }
}

```



**//WAP to create a tree.**

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct btnode
{
    int value;
    struct btnode *l;
    struct btnode *r;
}
*root=NULL;
void printout(struct btnode*);
struct btnode* newnode(int);
void main()
{
    root=newnode(50);
    root->l=newnode(20);
    root->r=newnode(30);
    root->l->l=newnode(70);
    root->l->r=newnode(80);
    root->r->r=newnode(60);
    root->l->l->l=newnode(10);
    root->l->l->r=newnode(40);
    printf("\n tree elements are\n");
    printf("\n display in order\n");
    printout(root);
    printf("\n");
    getch();
}
struct btnode* newnode(int value)
{
    struct btnode* node=(struct btnode*)malloc (sizeof(struct btnode));
    node->value=value;
    node->l=NULL;
    node->r=NULL;
    return(node);
}
void printout(struct btnode *tree)
{
    if (tree->l)
        printout(tree->l);
    printf("%d",tree->value);
    if(tree->r)
        printout(tree->r);
}

```