HANDWRITTEN NOTES

FREE DOWNLOAD

Data Structure & Algorithms

In C Programming

WWW.LEARNLONER.COM

DATA STRUCTURES

AND

ALGORITHMS.

Beginner to Advanced

GUIDE.

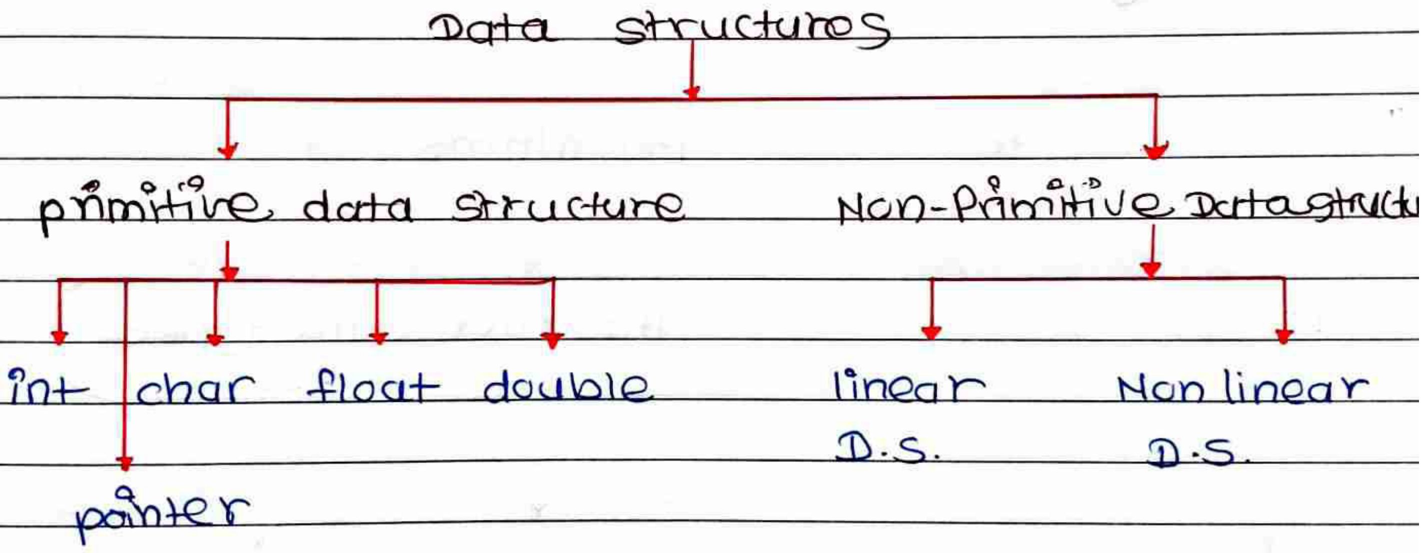
INDEX

What is Data Structure ?

Data structure is a way to store and organize data so that it can be used efficiently.

As per name indicates itself that organizing the data in memory.

The data structure is not any programming language like c, c++, Java etc. It is set of algorithms that we can use in any programming language to structure data in memory.



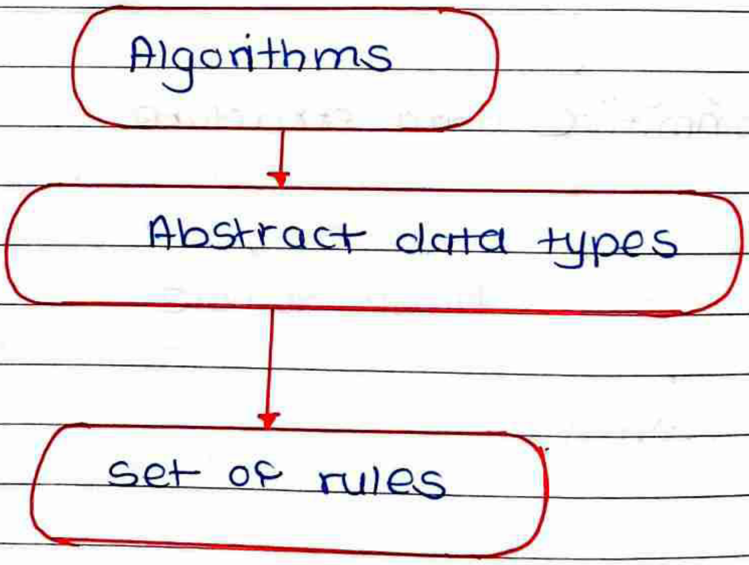
Linear Data structure :-

The arrangement of data in the sequential manner is known as linear data structu The data structure used for this purpose are Arrays, linked list, stacks and queues.

In this data structures, one element is connected to only one another element in a linear form.

Non-linear data structure :-
When one element is connected to the 'n' number of elements known as non-linear data structures.
Example :- trees and graphs.
In this case, elements are arranged in a random manner.

Algorithms and Abstract Data types ??



Why →

TO structure the data in memory, 'n' number of algorithms are proposed, and all these algorithms are knowns as Abstract Data Types.

An Abstract Data Type tells what is to be done and data structure tells how is to be done ?

ADT gives us the blueprint while data structure provides the implementation part.

What is Data ?

Data can be defined as the elementary value / collection of values.
forexample :- student's name and its id are the data about student.

What is Record ?

Record can be defined as collection of various data items
example :- student entity → name, address, course and marks can be grouped together to form record.

What is File ?

File is a collection of various records of one type of entity
example :- if there are 60 employees in class, then there will be 20 records in related file where record contains info of employee

What is Attribute and Entity ?

An entity represents class of certain objects. it contains various attributes. each attribute represents particular property of that entity.

What is need of data structures?

As applications are getting complexed and amount of data is increasing day by day, there may arrise following problems :-

Processor speed :- As data is growing day by day to the billions of files per entity, processor may fail to deal with that amount of data.

Data Structure :- consider an inventory size of 106 items in store, if our application needs to search for a particular item, it needs to transverse 106 items every time, results in slowing down process

multiple requests :- If thousands of users are searching data simultaneously on a web server, then there are chances that to be failed to search during that process.

To solve this problems, data structures are used. Data is organized to form a data structure in a such way that all items are not required to be searched and require data can be searched instantly.

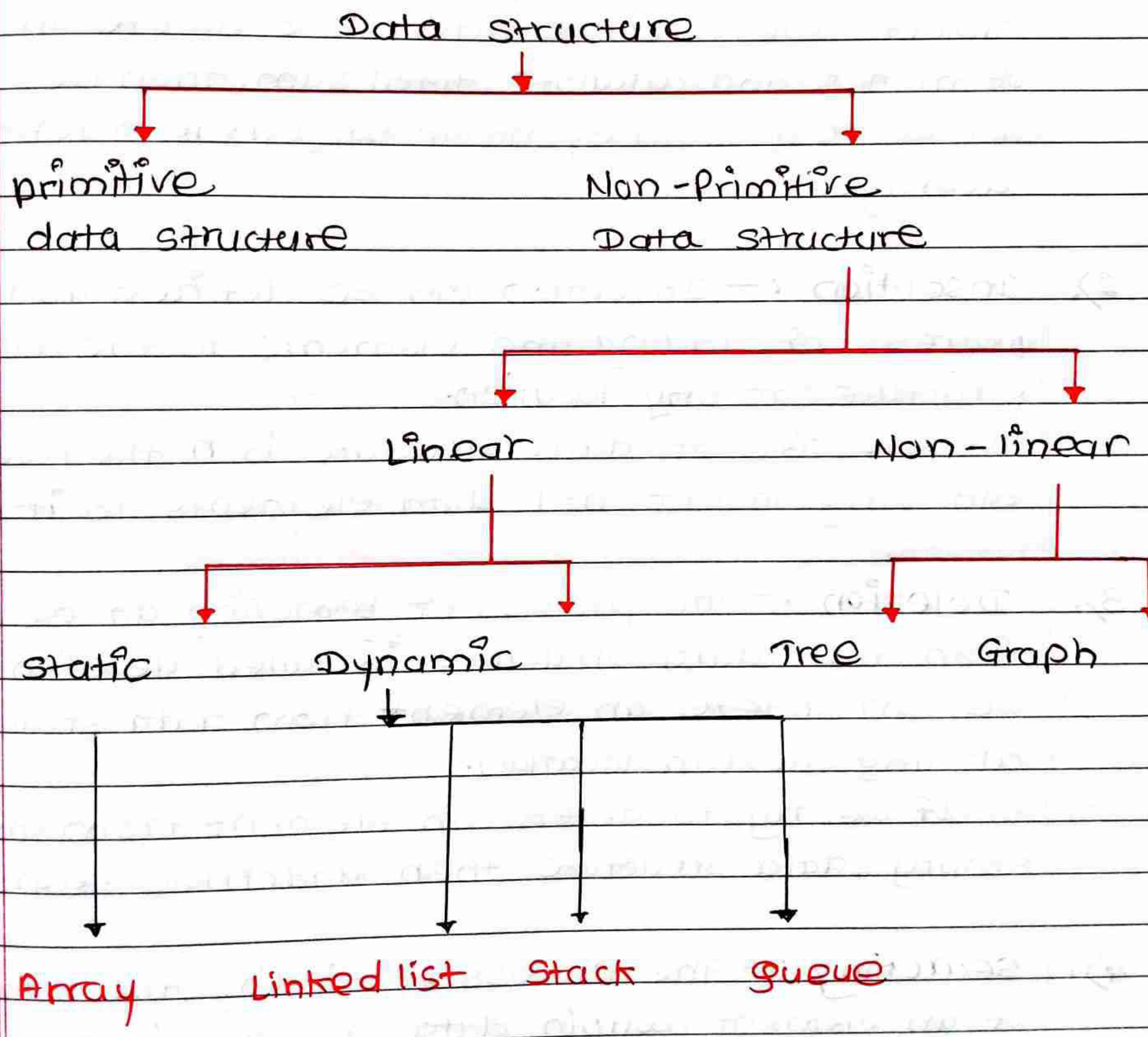
Advantages of data structure :-

Effiency :- If the choice of a data structure for implementing a particular ADT is proper, it makes program very efficient in terms of time and space.

Reusability :- The data structure provides reusability means that multiple client programs can use the data structure.

Abstraction :- The data structure specified by the ADT also provides level of abstraction. The client cannot see interval working of data structure, so it does not have to worry about implementation.

* Data structure classification :-



Operations on data structure :-

1). Traversing :- Every data structure contains a set of data elements. Traversing data structure means visiting each element of data structure in order to perform some specific operation like searching or sorting.

Example :- If we need to calculate average of marks obtained by a student in 6 different subject, we need to traverse complete array of marks and calculate total sum, then we will devide that sum by no. of subjects ie. 6 to find average.

2). Insertion :- Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is n then we can only insert n-1 data elements to it.

3). Deletion :- The process of removing an element from the data structure is called deletion. we can delete an element from data structure at any random location.

If we try to delete an element from an empty data structure then underflow occurs.

4). searching :- The process of finding the location of an element within data structure is called searching. There are two algorithms to perform searching, linear search and Binary search.

5). Sorting :- The process of arranging the data structure in a specific order is called as sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort etc.

6). merging :- When two lists list A and list B of size M and N respectively, of similar type of elements, clubbed or joined to produce third list, list C of size (M+N), then this process is called merging.

DATA STRUCTURES AND ALGORITHM

What is Algorithm?

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer.

It is not complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a flowchart or pseudocode.

characteristics of an algorithm.

Input :- An algorithm has some input values. We can pass 0 or some input value to an algorithm.

Output :- We will get 1 / more output at end of an algorithm.

Unambiguity :- An algorithm should be unambigous which means that instruction in an algorithm should be clear and simple.

Finiteness :- An algorithm should have finiteness. means limited number of instructions.

Effectiveness :- An algorithm should have finite as each instruction in an algorithm affects the overall process.

Approches in Algorithm :-

1). Brute force Algorithm :- The general logic structure is applied to design an algorithm. It is also known as exhaustive search algorithm that searches all possible to provide required solution.

such algorithms have two types :-

1). optimizing
finding all solutions of a problem and then take out the best solution is known then it will terminate if the best solution is known.

2). sacrificing
As soon as the best solution is found, then it will stop.

Divide and conquer :- This breaks down the algorithm to solve the problem in different methods. It allows you to break down problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.

Greedy algorithm :- It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting best solution. It is easy to implement and has faster execution time. But there are very rare cases in which it provides the optimal solution.

The major categories of algorithms are given below:

Sort :- Algorithm developed for sorting the items in a certain order.

Search :- Algorithm developed for searching the items inside a data structure.

Delete :- Algorithm developed for deleting the existing element from the data structure.

Insert :- Algorithm developed for inserting an item inside a data structure.

Update :- Algorithm developed for updating the existing element inside a data structure.

Algorithm Analysis :-

The algorithm can be analyzed in two levels i.e first is before creating the algorithm, and second is after creating the algorithm.

There are two analysis of an algorithm.

Prior Analysis :-

Here, prior analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm.

Posterior Analysis :-

Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing algorithm using any programming language.

Algorithm complexity :-

The performance of the algorithm can be measured in two factors :

Time complexity :-

The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation.

Here big O notation is the asymptotic notation to represent time complexity. The time complexity is mainly calculated by counting the number of steps to finish execution.

```
sum = 0 ;
// suppose we have to calculate the sum of n
 numbers.
for i=1 to n
sum = sum + i ;
// when the loop ends then sum holds the sum
of n numbers.
return sum;
```

In above code, the time complexity of the loop statement will be atleast n, and if value of n increases, then time complexity also increases.

We generally consider the worst-time complexity as it is maximum time taken for any given input size.

Space complexity :-

An algorithm's space complexity is the amount of space required to solve a problem and produce an output. simillar to the time complexity, space complexity is also expresed in big O notation.

Space complexity = Auxiliary space + Input size.

The following are the types of algorithms :

Search Algorithm :-

on each day, we search for something in our day to day life.

similarly, with the case of computer, huge data is stored in a computer that whenever user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search data in an array :

- Linear search
- Binary search

sorting Algorithms :-

sorting algorithms are used to rearrange elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements :

Asymptotic Analysis :-

The time required by an algorithm comes under three types :

Worst case :- It defines the input for which the algorithm takes a huge time.

Average case :- It takes average time for the program execution.

Best case :- It defines the input for which the algorithm takes the lowest time.

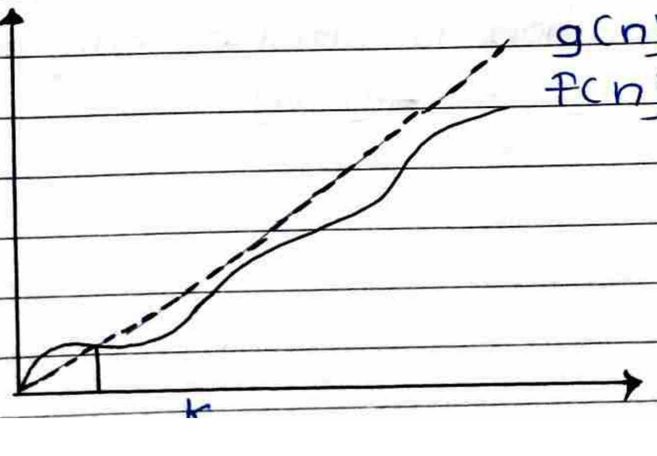
Asymptotic Notations :-

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below :

1). Big oh notation (O) :-

This measures the performance of an algorithm by simply providing the order of growth of the function.

This notation provides an upper bound on a function which ensures that function never grows faster than the upper bound.



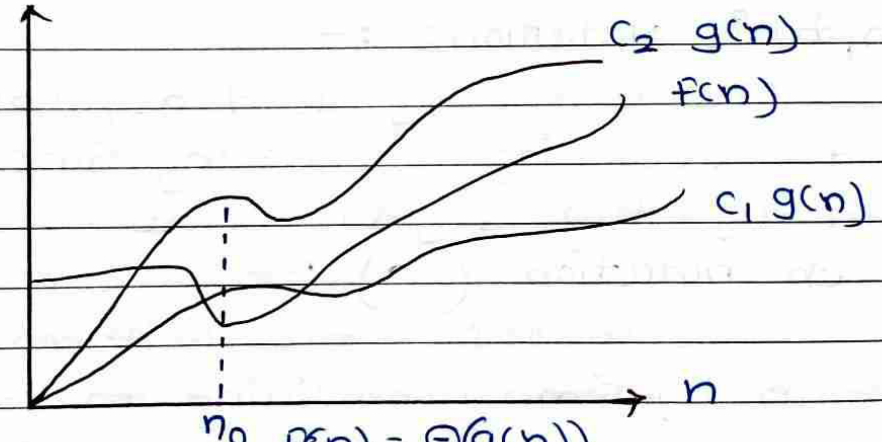
Example :- If f(n) and g(n) are two functions defined for positive integer,

then $f(n) = O\ g(n)$ as f(n) is big oh of g(n) or f(n) is on order of g(n)) if there exists constants c and n_0 such that :

$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

2). Omega Notation (Ω) :-

It basically describes best case scenario which is opposite to big o notation. It is the formal way to represent lower bound of an algorithm's running time.



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

Example :- let f(n) and g(n) be functions of n where n is steps required to execute program:

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

The above condition is satisfied only if when:

$c1.\ g(n) <= f(n) <= c2.g(n)$

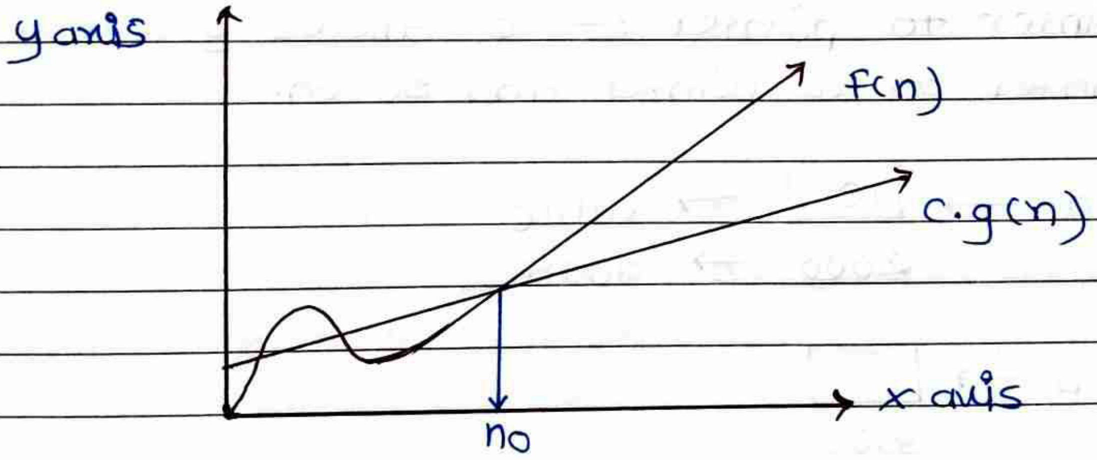
2). omega Notation (Ω)

It basically describes best-case scenario which is opposite to big-O notation It is formal way to represent lower bound to an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or best case time complexity.

Example :- If f(n) and g(n) are two functions defined for positive integers,

then $f(n) = \Omega\, g(n)$ as f(n) is omega of g(n) or f(n) is on the order of g(n) if there exists constants c and no such that :

$f(n) >= c \cdot g(n)$ for all $n \geq n_0$ and $c>0$



3). Theta Notation (Θ)

The theta notation mainly describes average case scenarios.

It represents realistic time complexity of an algorithm. Big theta is mainly used when the value of worst-case and best case is same.

Pointer :-

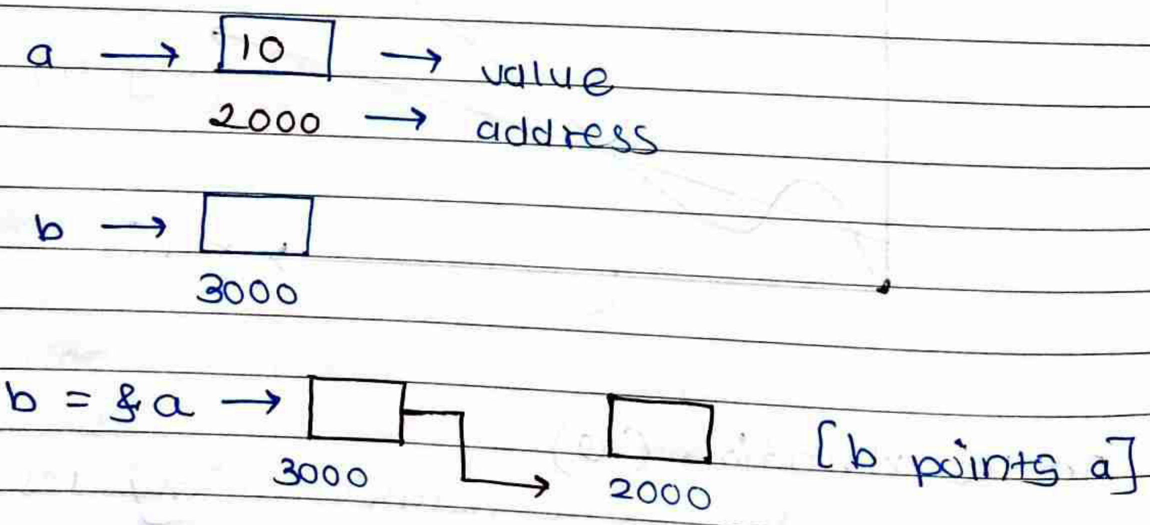
Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at location is known as dereferencing pointer.

Pointer arithmatic :-

4 arithmatic operators that can be used in pointers : ++, --, +, -

Array of pointers :- You can define array of to hold a number of pointers.

Pointer to pointer :- C allows you to have pointer on a pointer and so on.



Program
Pointer →

```
#include < stdio.h>
int main ()
```

```
{
 int a = 5 ;
 int *b ;
 b = &a ;
 printf ("value of a = %d \n", a);
 printf ("value of a = %d \n", *(&a));
 printf ("value of a = %d \n", *b);
 printf ("address of a = %u \n", &a) ;
 printf ("address of a = %d \n", b);
 printf ("address of b = %u \n", &b)
 printf ("value of b = address of a = %u", b);
 return 0 ;
}
```

output →

```
 value of a = 5
 value of a = 5
 address of a = 3010494292
 address of a = -1284473004
 address of b = 3010494296
 value of b = address of a = 3010494292.
```

Program :—
pointer to pointer :—

```
#include < stdio.h>
int main ()
 {
  int a = 5 ;
  int *b ;
  int **c ;
b = &a;
c = &b;
printf ("value of a = %d \n", a);
printf ("value of b = address of a = %u \n", b);
printf ("value of c = address of b = %u \n", c);
printf ("address of b= %u \n", c);
printf ("address of c = %u \n", &c);
return 0;
}
```

output →

```
value of a = 5
value of b = address of a = 2831685116
value of c = address of b = 2831685120
address of b = 2831685120
address of c = 2831685128.
```

Structure :-

A **structure** is a composite data type that defines a grouped list of variables that are to be placed under one name in block of memory.

Program :-
structure →

```
struct structure-name
{
 data-type member 1;
 data-type member 2;
```

```
  :
  :
  :
 data-type member ;
};
```

Advantages of structure :-

- It can hold variables of different data types.
- We can create objects containing different types of attributes.
- It allows us to re-use the data layout across programs.
- It is used to implement other data structure like linked list, queues, trees and graphs.

Program :-

how to use structure in program →

```
#include <stdio.h>
#include <conio.h>
void main ()
{
struct employee
{
int id ;
float salary;
int mobile;
};
```

```
  struct employee e1, e2, e3 ;
printf ("\n Enter ids, salary & mobile no. \n") ;
scanf (" %d %f %d", &e1.id, &e1.salary, &e1.mobile);
scanf ("%d %f %d ", &e2. id, &e2.salary, &e2.mobile);
printf(%d %f %d", &e3.id, &e3. salary, &e3.mobile);
printf("\n Entered result ") ;
printf ("\n %d %f %d", e1. id, e1. salary, e1.mobile);
printf ("\n %d %f %d ", e2.id, e2.salary, e2. mobile);
printf ("\n %d %f %d ", e3.id, e3.salary, e3.mobile);
getch () ;
}
```

output →   guess the output

And write it here....

Array :- Arrays are defined as collection of similar type of data items stored at contigous memory locations.

Array is the simplest data structure where each data element can be randomly accessed by using its index number.

Array declaration :-

```
int arr [10] ; char arr [10] ; float arr [5]
```

Program without Array :-

```
#include <stdio.h>
 void main ()
 {
  int marks_1 = 56; marks_2 =78, marks_3 = 89 ;
  Float avg = (marks_1 + marks_2 + marks_3)/3;
  print (avg);
 }
```

Program by using Array :-

```
#include <stdio.h>
void main
 {
  int marks [3] = {56, 78, 89};
  int i;
  float avg;
  for (i=0;i<3;i++)
```

```
        {
        avg = avg + marks [i];
        }
    printf (avg);
}
```

Complexity of Array operations :-

1). Time complexity :-

Algorithm	Arerage case	worst case
Access	O(1)	O(1)
search	O(n)	O(n)
insertion	O(n)	O(n)
Deletion	O(n)	O(n)

2). space complexity :-

In Array space complexity for worst case is O(n)

Memory Allocation of the Array :-

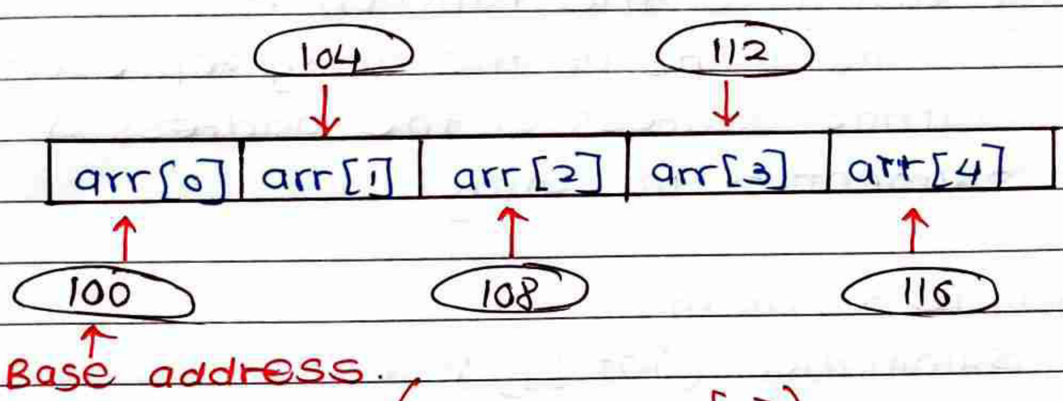
Each element in Array represented by indexing Indexing of array can be defined in three ways:

1. O (zero Based indexing) :-

The first element of the array will be arr [0].

2. 1 (one-based indexing) :-
The first element of array will be arr[1].

3. n (n-based indexing) :-
The first element of array can reside at any random index number.



(fig : int arr [5])

Accessing elements of an Array :-
To access any random element of an array we need the following information :
1. Base address of the array
2. size of an element in bytes.
3. Which type of indexing, array follows.

Address of any element of 1 D array can be calculate

Byte address of element A[i] = base address + size * (first - index)

Example: In an array, A[-10 ... +2] Base address (BA) = 999, size of an element = 2 bytes, find location of A[-1].

solution: $L(A[-1]) = 999 + \{(-1)-(-10)] \times 2.$

$= 999 + 18$

$= 1017.$

∴ location of $A[-1] = 1017.$

Passing array to the function :-

The name of the array represents the starting address or the address of the first element of the array.

Program:

```
#include <stdio.h>
int summation (int []);
void main ()
 {
  int arr [5] = {0,1,2,3,4};
  int sum = summation (arr);
  printf ("%d", sum);
 }
int summation (int arr[])
  {
   int sum =0, i;
   for(i=0 ;i<5 ;i++)
     {
      sum = sum + arr [i];
     }
   return sum ;
  }
```

2D Array :- 2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as collection of rows and coloumns.

How to declare 2D Array :-

The syntan for declaration of two dimensional array is as follows :

int arr [max - rows][max - coloumns];

However, it produces the data structure which looks like following :

	0	1	2	...	n-1
0	a[0][0]	a[0][1]	a[0][2]	.....	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	....	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	...	a[2][n-1]
⋮	⋮	⋮	⋮	...	⋮
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]		a[n-1][n-1]

a[n][n]

(Fig : a[n] [n])

How to access data in 2D-array :-

Due to fact that elements of 2D arrays can be random accessed.

```
int x = a[i][j];
```

where i, j are the rows and coloumns respectively.

Initializing 2D arrays :-

The syntax to declare and initialize the 2D array is given as follows :

```
int arr[2][2] = {0,1,2,3};
```

number of elements in 2D arrays
= number of rows * number of coloumns.

Mapping 2D array to 1 D array :-

The size of a two dimensional array is equal to the multiplication of number of rows and number of coloumns present in the array.

A 3x3 two dimensional array is a shown:-

	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

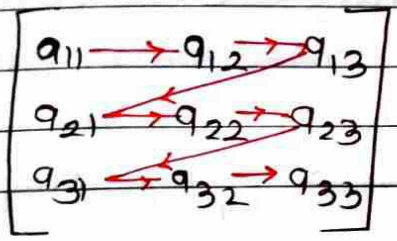
coloumn index

row index

There are two main techniques of storing 2D array elements into memory.

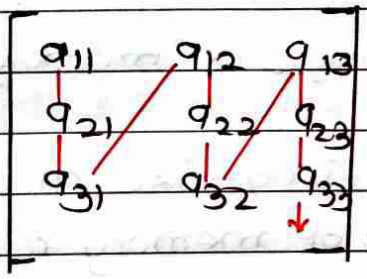
1. Raw major ordering :-

In row major ordering, all the rows of 2D array are stored into memory contiguously.



2. Colamn major ordering :-

According to coloumn major ordering, all the coloumns of 2D array are stored into the memory contigously.



calculating address of random element of a 2D array:-

1). By row major order :-

If array is declared a[m][n] where m is the number of rows while n is number of coloumns. then address of an element a[i][j] is calculated as,

Address (a[i][j]) = B.A + (i * n + j) * size

B.A → Base Address

2). By coloumn major order :-

Address (a[i][j] = (j * m) + i) * size + B.A.

Linked list :-

✓ why there is a need of linked list ?

If we declare an array of size 3. As we know that all the values of an array are stored in a continous manner, so all three values of an array are stored in a sequential fashion.

Then, total memory space occupied by array would be
3 * 4 = 12 bytes.

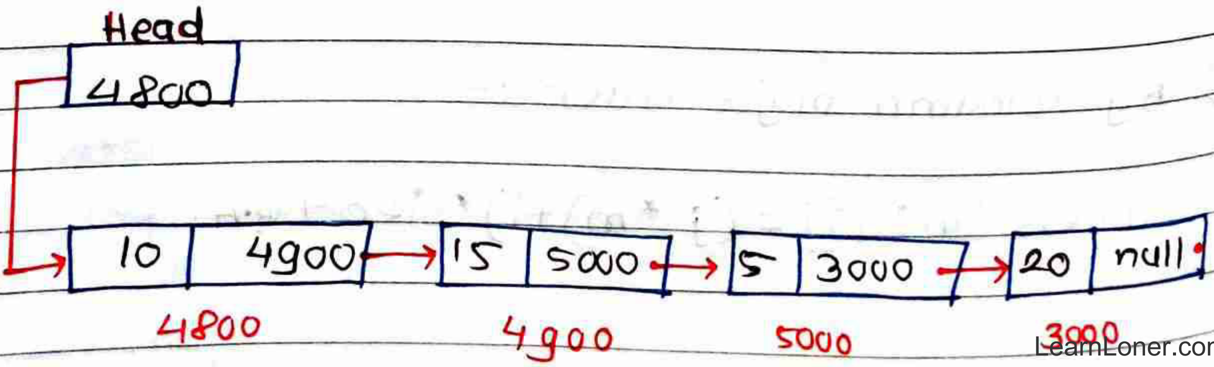
Drawbacks of using array :-

- we cannot insert more than 3 elements in above example because only 3 spaces are allocated by 3 elements.
- In case of array, the wastage of memory can occur.
- In array, we are providing fixed-size at compile time, due to which wastage of memory occurs.

The solution to this problem is to use linked list.

What is Linked List ?

A linked list is also a collection of elements, but the elements are not stored in a consecutive location. or linked list is a collection of the nodes in which one node is connected to another node and node consists of two parts i.e. one is data part and second one is the address part.



declaration of linked list :-

In linked list, one is variable and second one is pointer variable. we can declare linked list by using user-defined data type called as structure.

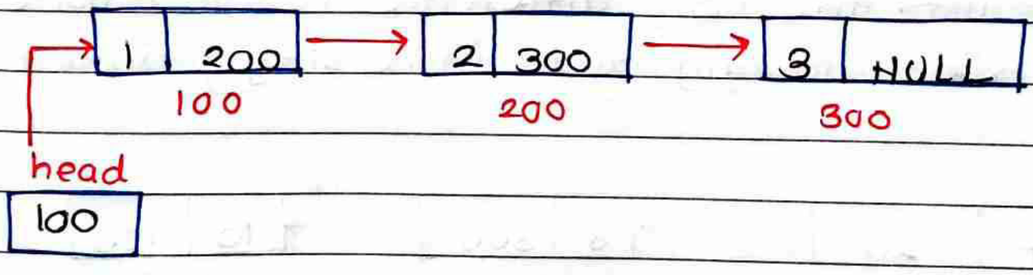
```
struct node
{
  int data;
  struct node *next;
}
```

Types of linked list :-

1). singly linked list :-

The singly linked list is most common. which consists of data part and address part. The address part in the node is known as a pointer.

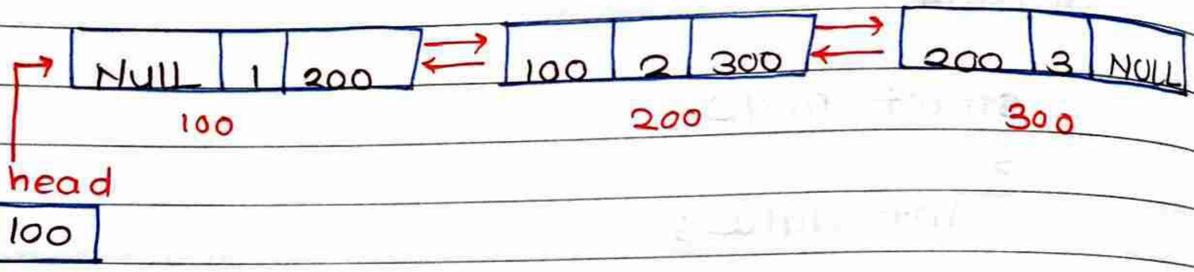
Example :- suppose we have three nodes and addresses of these three nodes are 100, 200 and 300 :



NULL means its address part does not point to any node. The pointer that holds the address of the initial node is known as a head pointer.

2. Doubly linked list :-

As name suggests, the doubly linked list contains two pointers. We define it in three parts the data part and the two address part.

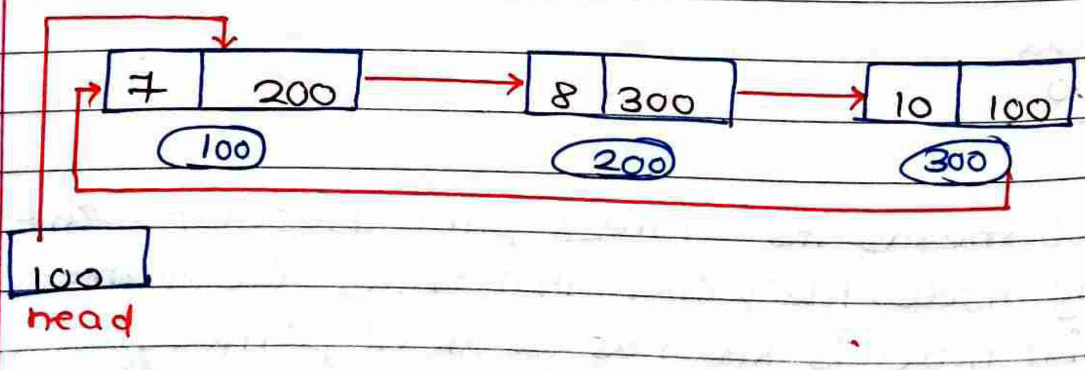


Representation of doubly linked list :-

```
struct node
 {
  int data ;
  struct node *next ;
  struct node * prev;
 }
```

3. Circular linked list :-

A circular linked list is a variation of a singly linked list. The only difference is "last node does not point to any node in a singly linked list".

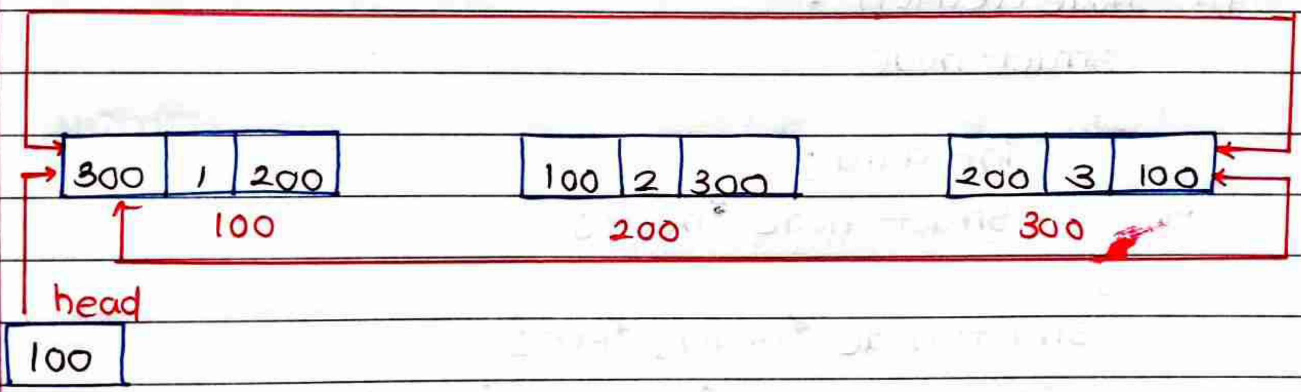


Representation of circular linked list :-

```
struct node
 {
   int data;
   struct node *nest;
 }
```

4. Doubly circular linked list :-

The doubly circular linked list has the features of both the circular linked list and doubly linked list.



The last node is attached to the first node and thus creates a circle.
The main difference is that doubly circular linked list does not contain NULL value in previous field of the node.

Representation of doubly circular linked list :-

```
struct node
 {
   int data;
   struct node *next;
   struct node *prev;
 }
```

complexity :-

	Average				Space complexity
Singly linked list	Access	search	Insertion	deletion	Worst
	O(n)	O(n)	O(1)	O(1)	O(n)
	Worst				
singly linked list	Access	search	Insertion	deletion	
	O(n)	O(n)	O(1)	O(1)	

Operations on singly linked list :-

1). Node creation :-

```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
ptr= (struct node *) malloc(sizeof (struct node *)
```

2). Insertion :-

① Insertion at beginning :- It involves inserting any element at the front of the list. We just need a few link adjustment to make new node as head of list.

②. Insertion at end of list :- The new node can be inserted as the only node in the list / it can be inserted as last one.

③. Insertion after specified node :- we need to skip desired number of nodes in order to reach node after which the new node will be inserted.

3). 3). Deletion and Traversing :-

①. Deletion at beginning :- It just needs few adjustments in the node pointers

② Deletion at end of list :- The list can either be empty or full. Different logic is implemented for different scenario's.

Traversing :- In traversing, we simply visit each node of the list at least once in order to perform some specific operation in it, for example, printing data part of each node present in the list.

Searching :- In searching, we match each element of the list with the given element. If the element is found on any of the location of that element is returned otherwise null is returned.

operations on doubly linked list :-

1). Node creation :-

```
struct node
 {
   struct node *prev;
   int data;
   struct node *next;
 };
struct node *head;
```

2). Insertion :-

①. Insertion at begining :- Adding the node into the linked list at beginning.

② Insertion at end :- Adding the node into the linked list to the end.

3). Deletion and Traversing :-

① Deletian at beginning :- Removing the node from beginning of the list.

②. Deletion at end :- Removing the node from end of the list.

Traversing :- visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display etc.

Searching :- compairing each node data with the item to be searched and return location of the item in the list if the item found else return null.

Skip list :-

* What is a skip list ?

A skip list is a probalistic data structure. The skip list is used to store a linked list of elements or data with a linked list. In one single step, it skips sereral elements of the entire list, which is why it is known as skip list.

Structure of skip list :-

skip list is built in two layers: The lowest layer and the top layer. The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are the like an "express line" where elements are skipped.

complexity table :-

Sr.No	complexity	Average case	Worst case
1).	Access complexity	o(logn)	o(n)
2).	search comple.	o(logn)	o(n)
3).	delete comple.	o(log n)	o(n)
4).	Insert comple.	o(logn)	o(n)
5).	space comple.	–	o(nlogn).

Basic operations and its algorithms :-

1). Insertion operation :- It is used to add new node to a particular location in a specific situation.

2). Deletion operation :- It is used to delete a node in a specific situation.

3). search operation :- The search operation is used to search a particular node in a skip list.

Algorithm of insertion operation :-

```
Insertion (L, key)
local update [0 ... max - level +1]
q = L → header
for i = L → level down to 0 do.
        while a → forward [i] → key forward [i]
update [i] = a
```

```
    a = a →forward[0]
    lvl = random-level()
    if lvφ > L →level then
    for i = L → level + 1 to lvl do
        update [i] = L → header
        L → level = lvl
    a = make node (lvl, key, value)
    for i = 0 to level do
       a → forward [i] = update [i] →forward [i]
    update [i] → forward [i] = a
```

Algorithm of deletion operation :-

```
Deletion (L, key)
local update [0.... man level + 1]
 a = L → header
for i = L → level down 0 to do.
   while a → forward [i] → key forward [i]
       update [i] = a
a = a → forward [0]
if a → key = key then
   for i = 0 to L → level do
   if update [i] → forward [i] ≠ a then break
   update [i] → forward [i] → forward [i]
 free(a)
while L → level > 0 and L → header → forward [L → level]
                                  = NIL do
  L → level = L → level - 1.
```

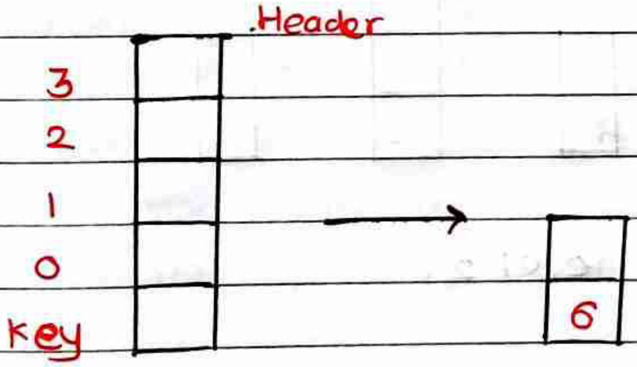
Algorithm of searching operation :-

```
searching (L, Skey)
    a = L → header
    loop invariant : a → key level down to 0 do.
        while a → forward [i] → key forward [i]
    a = a → forward [a]
    if a → key = skey then return a → value
    else return failure.
```

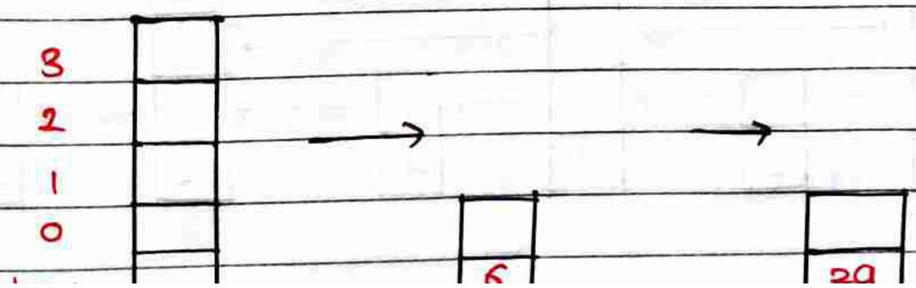
Example : create a skip list, we want to insert these following keys in empty skip list.

1. 6 with level 1
2. 29 with level 1
3. 22 with level 4.
4. 9 with level 3.
5. 17 with level 1.
6. 4 with level 2.

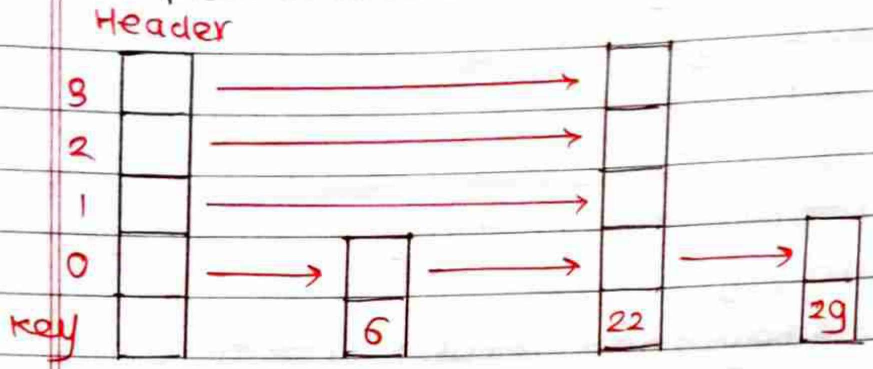
→ Solution :- Insert 6 with level 1.



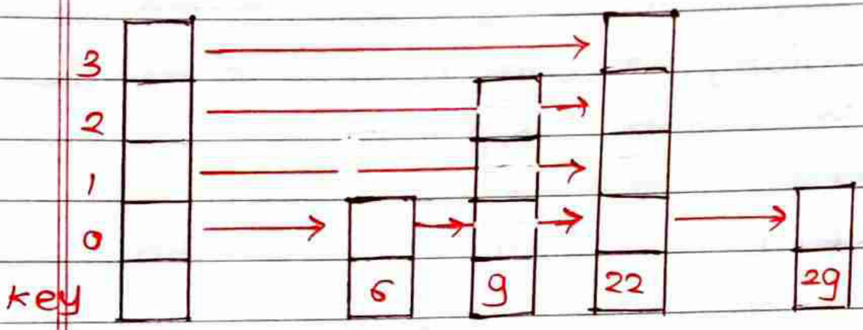
step 2 :- Insert 29 with level 1.



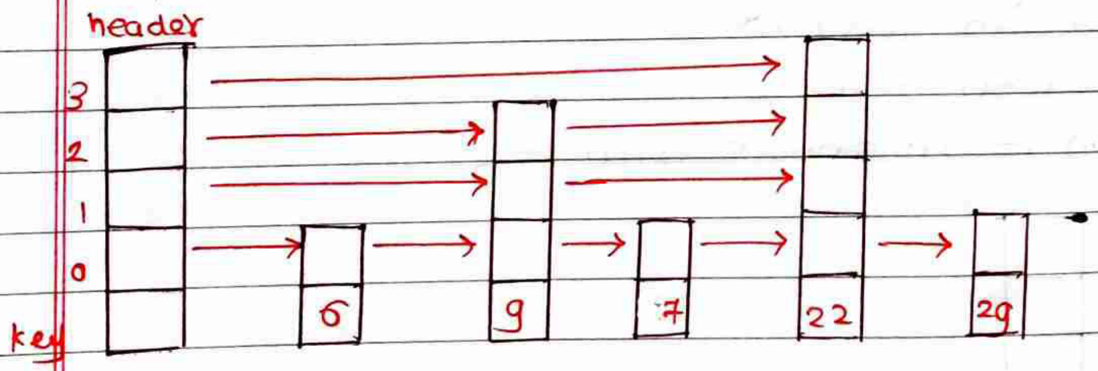
Step3 : Insert 22 with level 4.



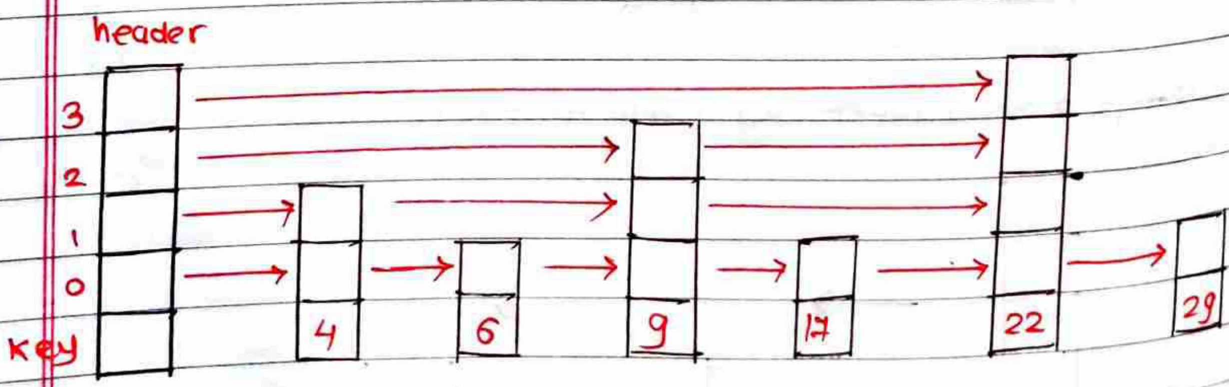
Step 4: Insert 9 with level 3.



Step 5: Insert 17 with level 1



Step 6: Insert 4 with level 2.



Stack :- A stack is a linear data structure that follows LIFO (Last-In-First-Out) principle. stack has one end, whereas queue has two ends (front and rear).

A stack is a container in which insertion and deletion can be done from the end (one) known as the top of the stack.

A stack is an Abstract Data Type with a pre-defined capacity, which means that it can store elements of limited size.

Operations on the stack :-

1). push ( ) :- When we insert an element in a stack then the operation is known as push. If stack is full overflow condition occurs.

2). pop ( ) :- when we delete an element from stack, the operation is called as pop ( ). If stack is empty means no element exists in the stack, this state is known as an underflow state.

3). peek ( ) :- It returns the element at a given position.

4). count ( ) :- It returns the total number of elements available in a stack.

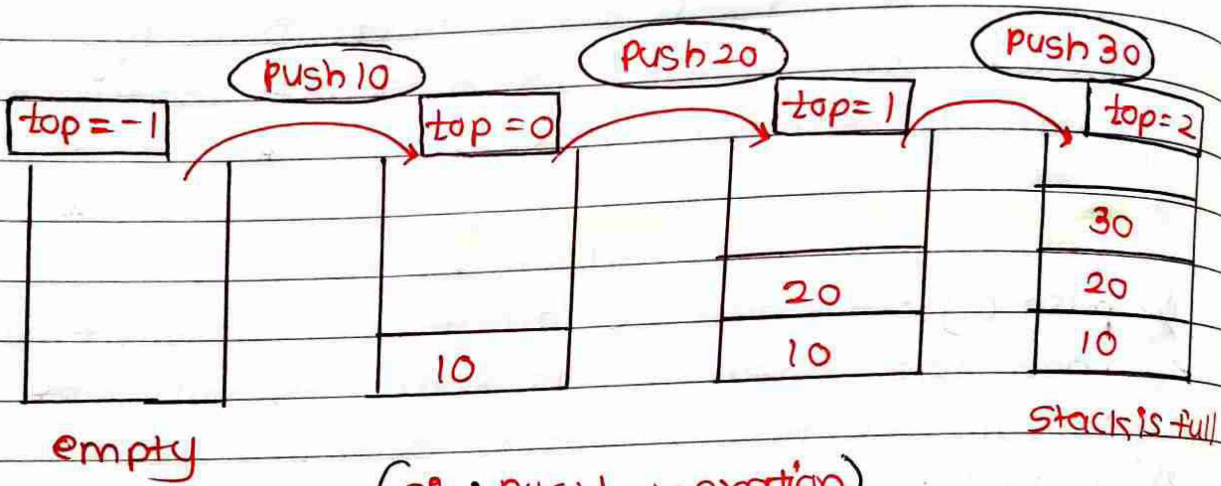
5). change ( ) :- It changes the element at the given position.

6). display ( ) :- It prints all the elements available in the stack.

PUSH operation :-

steps - Before inserting an element in the a stack, we check whethere the stack is full.

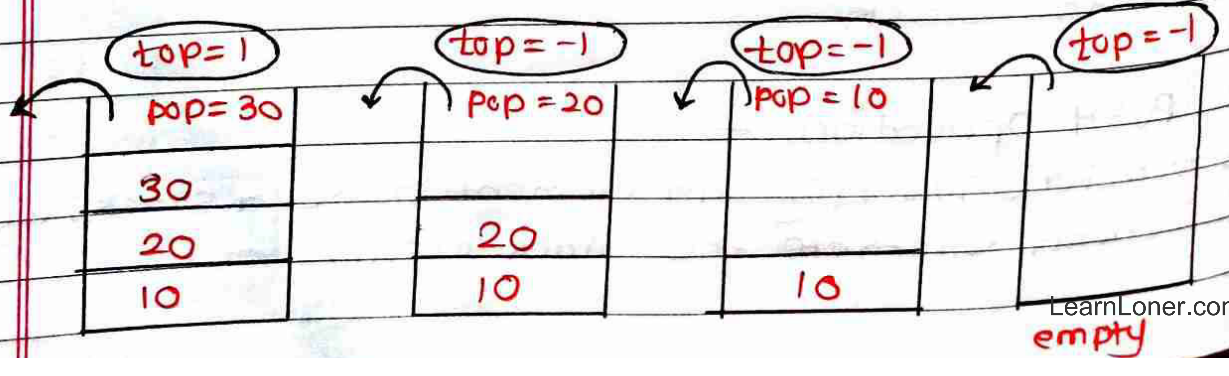
- If we try to insert element in a stack, and the stack is full, then overflow condition occurs.
- when we initialized a stacks, we set the value of top as -1 to check that stack is empty.
- The elements will be inserted until we reach the max size of the stack. top = top+1.



(fig: PUSH operation)

POP Operation :-

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from empty stack, then underflow condition occurs.
- first access the element which is pointed by top.
- once the top operation is performed, top is decremented by 1 i.e. top = top-1.



Applications of stack :-

1). Recursion :- The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all previous records of function are maintained.

2). DFS (Depth first search) :- This search is implemented on a graph, graph uses stack d.s.

3). Backtracking :- If we have to create a path to solve maze problem, If we are moving in particular path and we realise that we come on the wrongway. In order to come at beginning of the path to create a new path, we use stack d.s.

4). memory management :- The stacks manages the memory. The memory is assign in the contiguous memory blocks.

Algo :- push operation :-

```
begin
   if top = n then stack full
   top = top + 1
   stack( top) : = item ;
end
```

Time co Complexity : O(1)

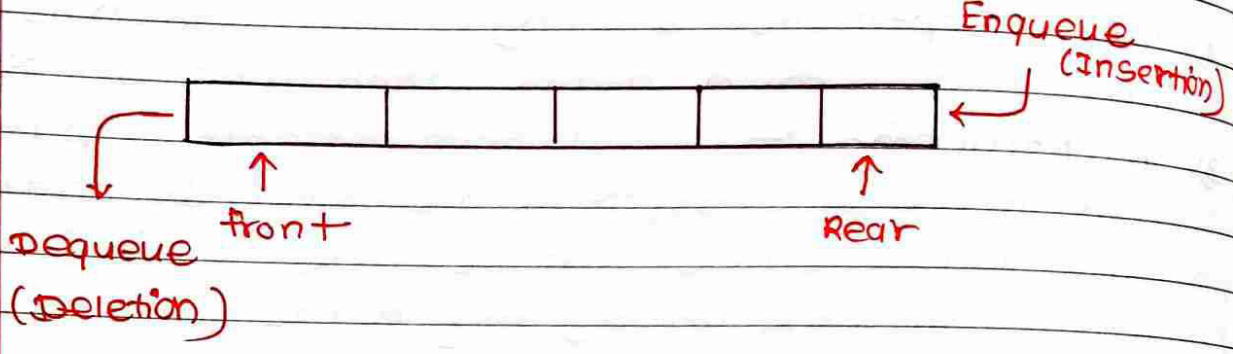
pop operation :-

```
begin
   if top = 0 then empty
   item : = stack ( top) ;
   top = top - 1 ;
end.
```

Time complexity : O(1)

Queue :- A queue can be defined as ordered list which enables insert operations to be performed atone end called REAR and delete operations to be performed at another end called FRONT.

- Queue can be referred as to be first In first out list.



Complexity of queue :-

	Average				Space comp
	Access	search	Deletion	Insertion	worst
Queue	O(n)	O(n)	O(1)	O(1)	O(n)
	Worst				
	Access	search	Insertion	Deletion	
Queue	o(n)	o(n)	o(1)	o(1)	

Operations on queue :-

1). Enqueue :- Enqueue is used to insert element at rear end of the queue. It returns void.

2). Dequeue :- dequeue operations performs the deletion from front end of queue. The deque operation can also be designed to void.

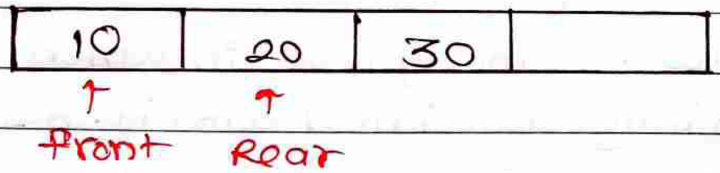
3). peek :- This returns, element which is pointed by front pointer in the queue but does not delete it.

4). queue overflow (is full) :- when queue is completely full, then it shows overflow condition.

5). queue underflow (isempty) :- when there is no element in the queue then it throws underflow condition.

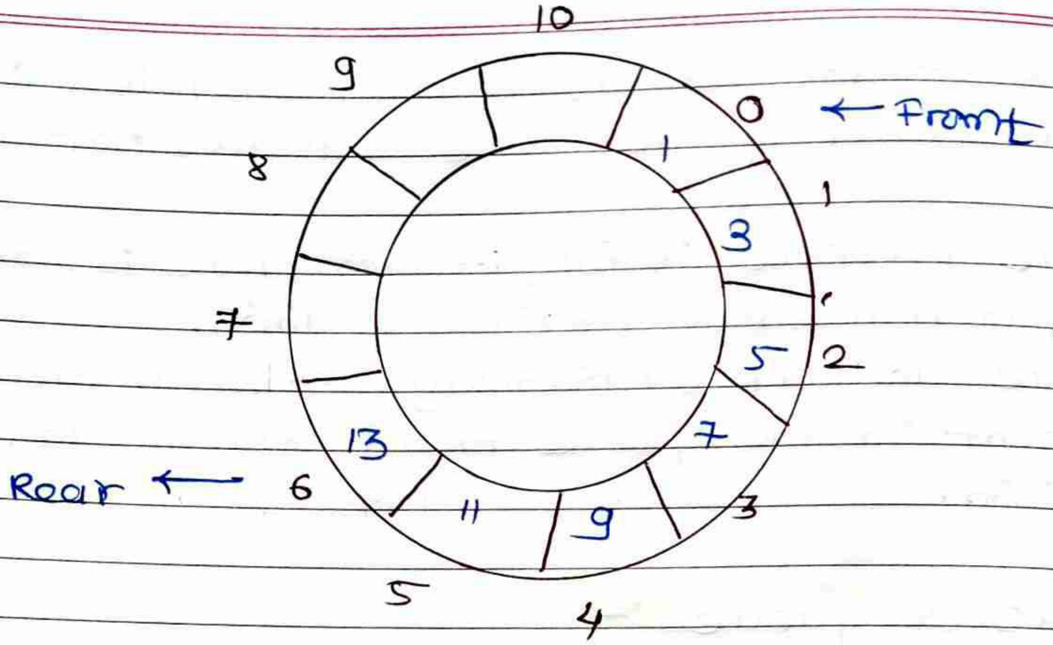
Types of queue :-

1). Linear queue :- In linear queue, an insertion takes place from one end while deletion occurs from another end. It strictly follows FIFO rule. The linear queue can be represented, as shown :



The elements are inserted from rear end, and if we insert more elements in queue, then rear values gets incremented on every insertion.
drawback is using linear queue is : insertion is done only from rear end. The linear queue shows the overflow condition as rear is pointing to last element of the queue.

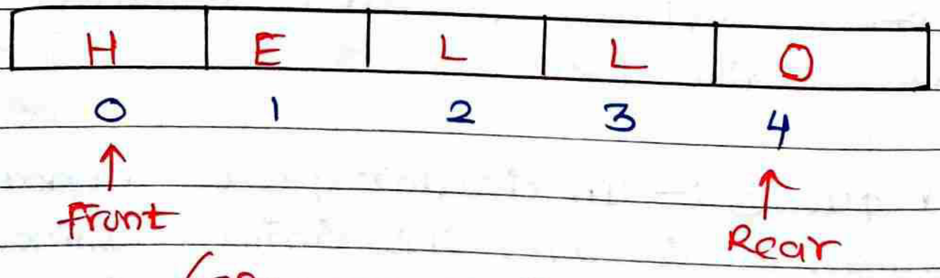
2). circular queue :- In circular queue, all nodes are represented as circular. It is similar to linear queue except that last element of the queue is connected to the first element. It is also known as ring buffer. as all ends are connected to another end.



Drawback of linear queue is overcomes in circular queue. If empty space is available, new element can be added in empty space by simply incrementing value of rear.

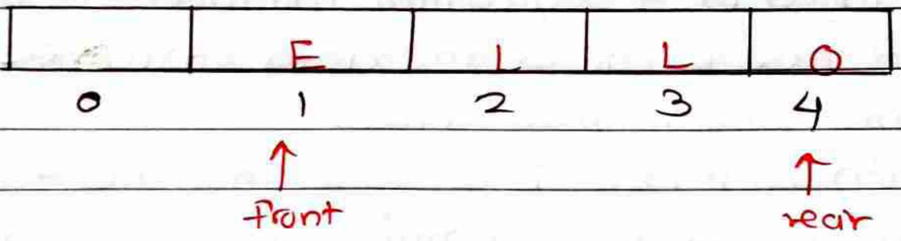
3). Priority queue :- The queue in which each element has some priority associated with it. Based on priority of the element, elements are arranged in a priority queue. If elements occur with same priority, then they are served according to FIFO principle.

* Array representation of queue :-



(Fig : queue after inserting an element)

After deleting element, value of front will increase from -1 to 0, the queue will look like :



(Fig : queue after deleting an element)

Algorithm to insert any element in a queue :-
check if queue is already full by comparing rear to max -1.

```
Algo: step 1 :- IF REAR = MAX -1
          write OVERFLOW
          Go to step  [END OF IF]
      step 2 :- IF FRONT = -1 and  REAR = -1
          SET FRONT = REAR = 0
          ELSE
          SET REAR = REAR +1   [END OF IF].
      step 3 :- SET QUEUE [REAR] = NUM
      step 4 :- EXIT.
```

Algorithm to delete an element from queue :-

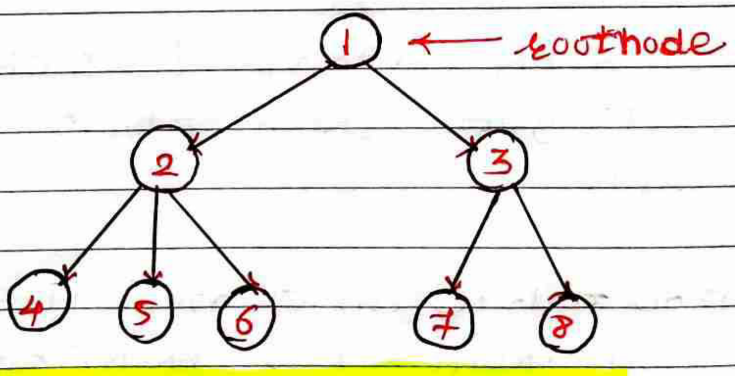
```
Algo: step 1 :- IF FRONT = -1 or FRONT > REAR
          write UNDERFLOW
          ELSE
          SET VAL = QUEUE [FRONT]
          SET FRONT = FRONT +1
          [END OF IF]
      step 2 :- EXIT.
```

Tree :- we read data structure, like an array, linked list, stack and queue in which all elements are arranged in a sequential manner.

A tree is one of the data structures that represents hierarchical data.

defination :- A tree is a data structure defined as collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy

- A tree is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in tree are arranged in multiple levels.
- In tree data structure topmost node is called as root node. Each node contains some data, & data can be of any type.
- Each node contains some data & links or reference of other nodes that can be called children.



Some Basic terms of tree :-

1). link :- each node is labeled with some number. each array shown in fig is known as links between two nodes.

2). Root :- The root node is topmost node in tree hierarchy. root node is one that doesn't have any parent. If node is directly linked to some other node, then it would be called a parent-child relation-ship.

3). child node :- If the node is a descendant of any node, then node is called as child node.

4). parent :- If node contains any sub-node, then node is said to be parent of that sub-node.

5). sibling :- The nodes that have same parents are called siblings.

6). leaf node :- node which doesn't have any child node, a leaf a bottom-most node of tree.

7). ancestor node :- It is any predessor node on a path from root to that node. In the given fig, 1, 2, 5 are ancestors of node 10.

8). Descendant :- The immediate successor of given node is known as descendant of a node.

* Properties of tree data structures :-

1). Recursive data structure :- Tree is also known as recursive data structure. Recursion means reducing something in a self-similar manner.

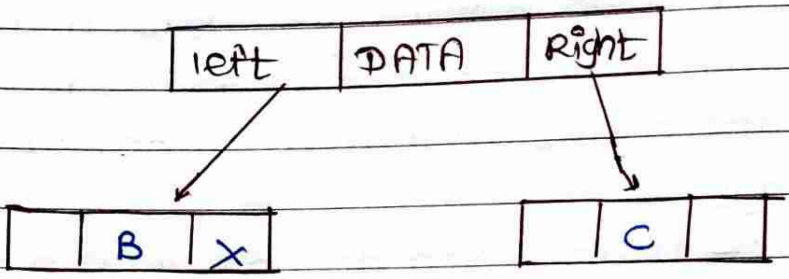
2). Number of edges :- If there are (n) nodes, then there would be (n-1) edges. each node, except root node, will have atleast one incoming link known as an edge.

3). Depth of node x :- It can be defined as length of path from root to node x. one edge contributes one unit length in the path, depth can be defined as no. of edges between root node and node (x)
The root node has depth 0.

4). Height of node x :- It is defined as longest path from node x to leaf node.

Impremontation of tree :-

The tree data structure can be created nodes dynamically with help of pointers. The tree in memory can be represented as shown :



```
struct node
 {
  int data;
 struct node *left;
  struct node *right;
 }
```

The above structure can only be defined for binary trees because binary tree can have utmost two children, and generic trees.

Application of trees :-

1). Storing naturally hierarchical data :- File system, stored on disc drive, file and folder are in form of naturally heirarchical data and store in form of trees.

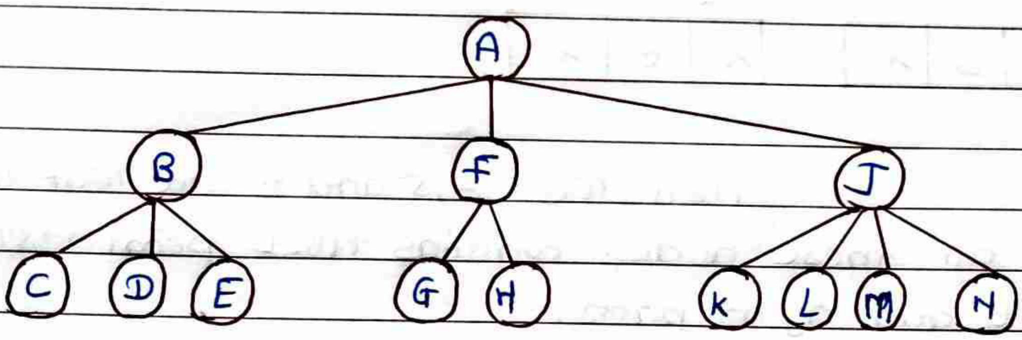
2). organize data :- It is used to organize data for efficient insertion, deletion and searching.

3). Trie :- It is special kind of tree that is used to store dictionary. It is fast and efficient way for dynamic spell checking.

4). Heap :- It is also a tree data structure implemented using arrays. It is used to implement priority queues.

Types of Tree data structure :-

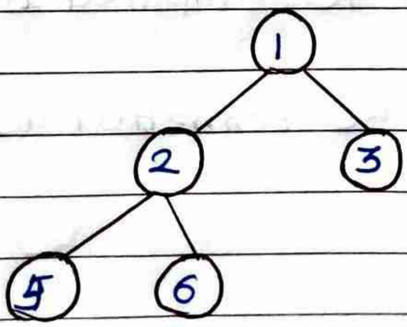
1). General Type :- In a general tree, a node can have either 0 or maximum n number of nodes. There is no restrictions imposed on the degree of node (number of nodes that a node can contain) The topmost node in a general tree is known as root node. The children of parent node are known as subtree.



There can be n number of subtrees in general tree. In general tree, subtrees are unordered as nodes in subtree cannot be ordered.

Every non empty tree has a downward edge, and these edges are connected to nodes known as child nodes. The nodes that have same parent are known as siblings.

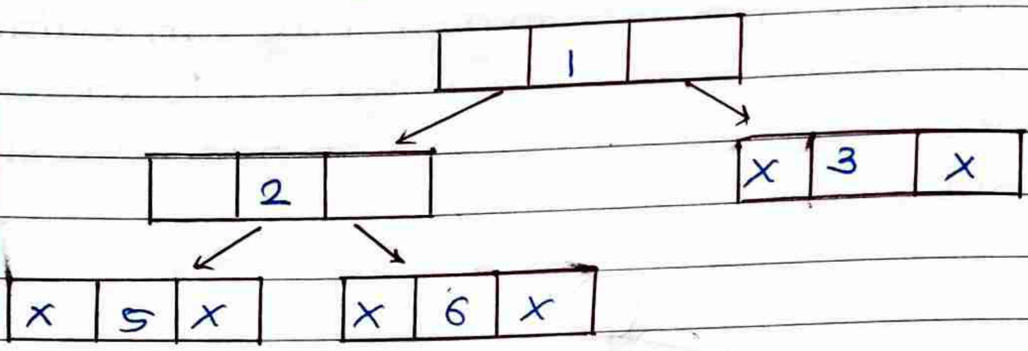
2). Binary Tree :- Binary tree means that the node can have maximum two children.



← given tree is a binary tree because
- each node contains atmost two children.

logical representation of above tree :-

In above tree, node 1 contains two pointers i.e left and right pointer pointing to left and right node respectively.



The nodes 3, 5 and 6 are leaf nodes, so all these nodes contains NULL pointer on both left and right parts.

Properties of Binary tree :-

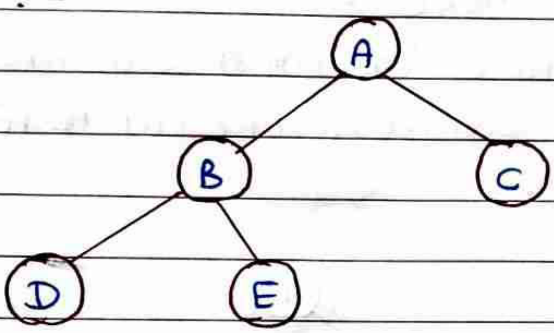
- At each level of i, the maximum number of nodes is 2^i.
- The height of tree is longest path from root node to leaf node. In general, maximum number of nodes possible at height is $(2^0+2^1+2^2+..2^h)$
- The minimum number of nodes possible at height h is equal to h+1.
- If number of nodes is minimum, then height of tree would be maximum.
- minimum height can be computed as:
  $h = \log_2(n+1)-1$
- maximum height can be computed as:
  $h = n-1$.

Types of Binary tree :-

1). full / proper / strict Binary tree :-

If each node contains either 0 or two children. The tree in which each node must contain 2 children except left nodes.

Example :-



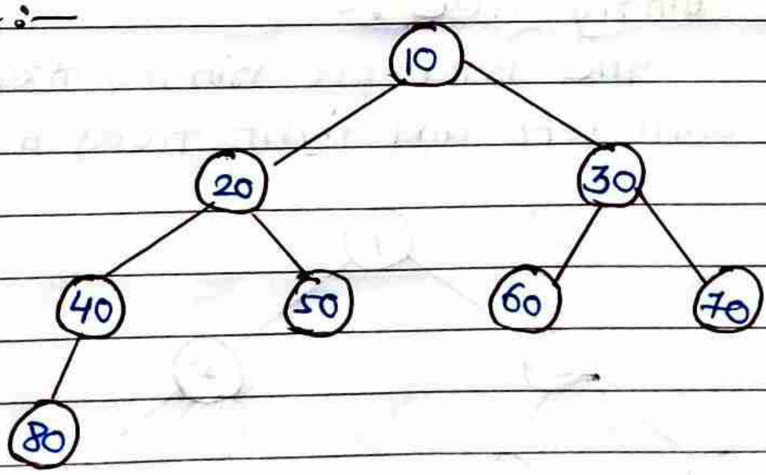
Properties :-

- maximum number of nodes : $2^{h+1}-1$.
- minimum number of nodes : $2*h-1$
- minimum height $\log_2(n+1)-1$
- maximum height $h = \frac{n+1}{2}$

2). complete Binary Tree :-

The tree in which all nodes are complete-ly filled except the last level. In complete Binary tree nodes should be added from left.

Example :-



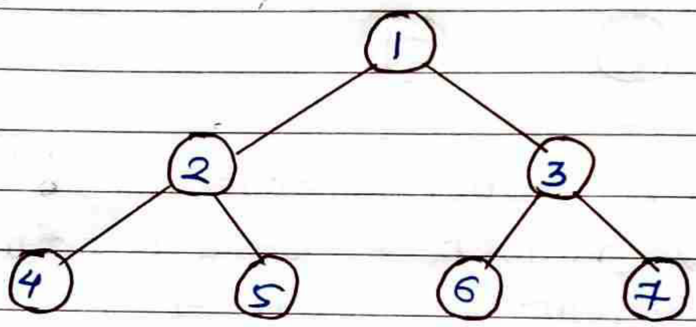
properties :-

- maximum number of nodes → $2^{h+1}-1$.
- minimum number of nodes → 2^h
- minimum height → $\log_2(n+1)-1$.

3). Perfect Binary tree :-

A tree in which all the internal nodes have 2 children. and all leaf nodes are at the same level.

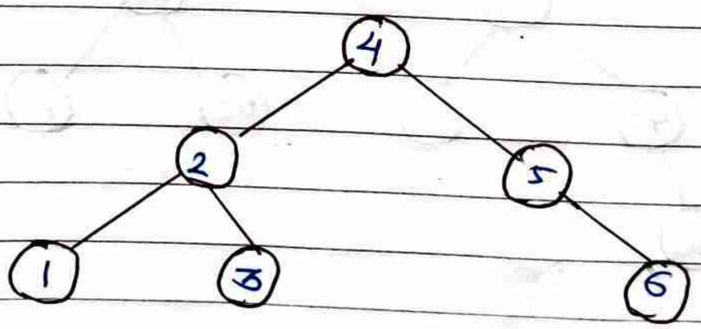
Example :-



Note :- All the perfect Binary trees are complete binary trees as well as the full Binary trees as But, vice versa is not true, all complete binary trees and full binary trees are the perfect Binary trees

4). Balanced Binary Tree :-

The balanced binary tree is a tree in which both left and right trees by almost 1.



above tree is balance : diff betn left subtree & right S.T. is zero

Binary Tree Implementation :-

```
struct node
 {
  int data ;
  struct node *left, *right ;
 }
```

Tree Traversal :-

The process of visiting nodes is called as tree traversal.

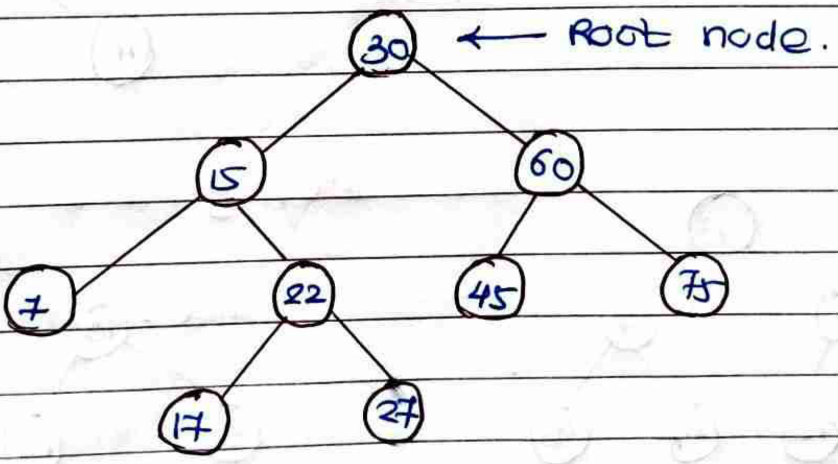
There are three types of traversals used to visit a node :

1). Inorder Traversal
2). preorder Traversal
3). postorder Traversal.

3). Binary Search Tree :-

defn :- Binary search tree can be defined as a class of binary trees, in which a nodes are arranged in a specific order. also called as ordered BinaryTree.

- similarly value of all nodes in right subtree is greater than or equal to value of root.



Example : create binary search tree using following data elements :-

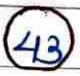
43, 10, 79, 90, 12, 54, 11, 9, 50.

→ 1). Insert 43 into tree as root of tree.

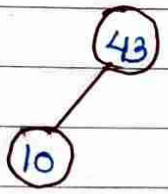
2). Read next element, if it is lesser root node element, insert it as root of left sub-tree.

3). otherwise, insert it as root of right of right subtree.

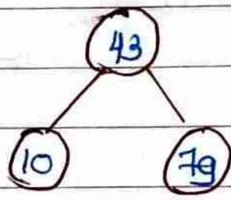
step 1



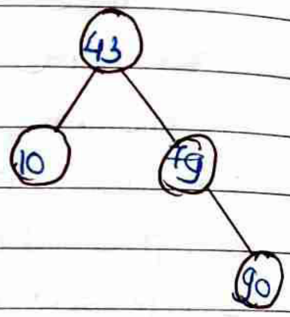
step 2



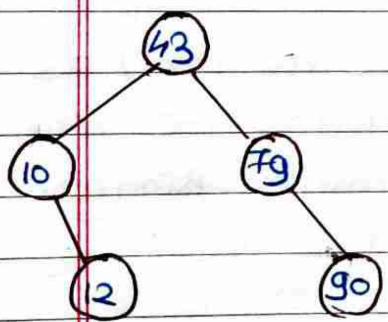
step 3



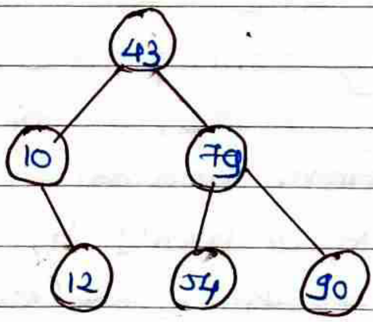
step 4



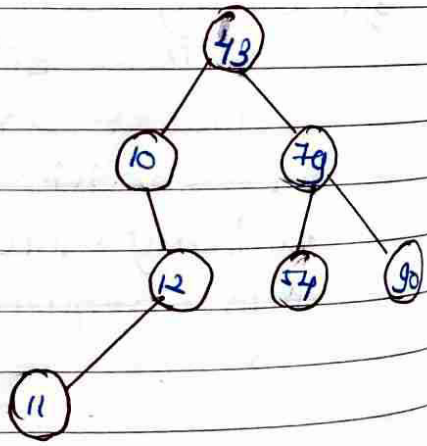
step 5



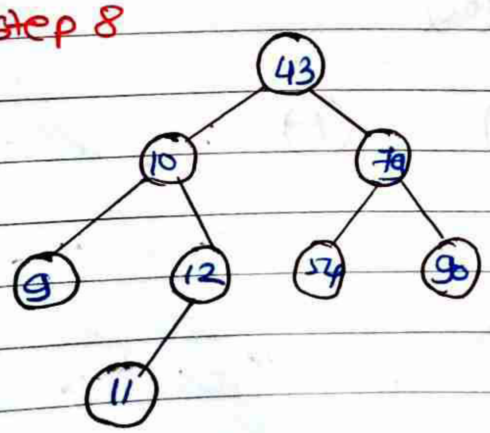
step 6



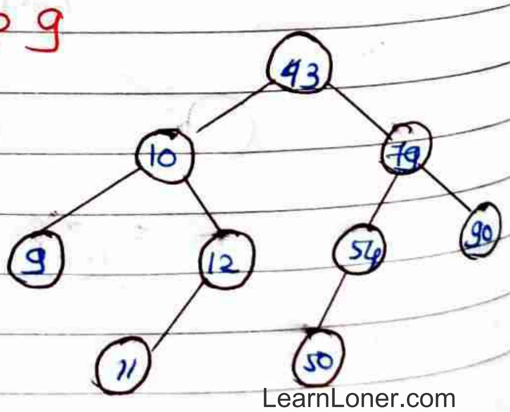
step 7



step 8



step 9



Operations on Binary Search Tree (BST) :-

Sr.No.	Operation	Description
1).	searching in BST	Finding location of some specific element in a Binary search Tree.
2).	Insertion in BST	Adding a new element to the binary search tree at appropria location so that property of BST do not violate.
3).	Deletion in BST.	Deleting some specific node from a BST, However, tree there can be various cases in deletion depending upon number of children, node have.

4). AVL Tree :- AVL Tree is invented by GM Adelson -velsky and EM Landis in 1962. The tree is named as AVL in honour of its inventors.

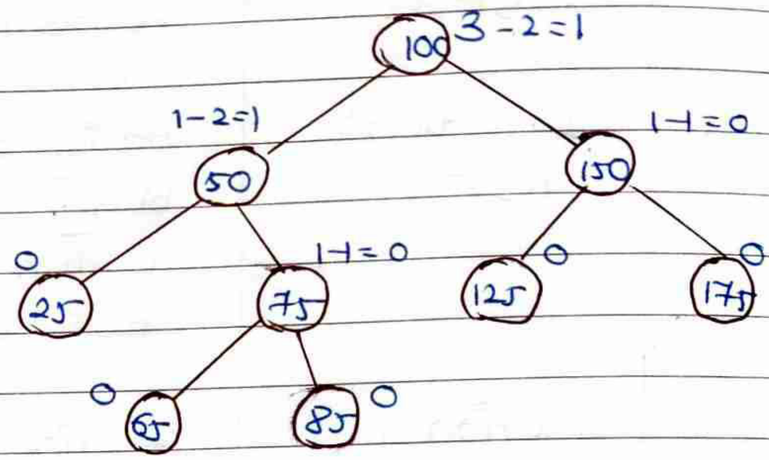
AVL tree is defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its R. subtree from its left subtree.

Balance Factor (K) = height (left (K)) − height (right (K))

- If balance factor of any node is 1, it means that left sub-tree is one level higher than right subtree.

- If balance factor of any node is 0, it means that left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that left sub-tree is one level lower than right subtree.

Example :-



Here we see that, balance factor associated with each node is between -1 and +1.
∴ It is an example of AVL tree.

complexity :-

Algorithm	Average case	Worst case
space	O(n)	O(n)
search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n).

Why AVL Tree ? → AVL tree controls height of binary search tree by not letting it to be skewed. The time taken by all operations in BST is O(h). However it will be extended to O(n) If BST became skewed (worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be O(logn), where n is number of nodes.

Operations on AVL tree :-

Sr. No	Operation	Description.
1).	Insertion	Insertion is performed in same way it performed in BST. However, it may lead to violation in the AVL tree property and so tree may need balancing. and tree can be balanced by rotation.
2).	Deletion	Deletion is also same way performed as BST It can be also disturb balance of tree, so various types of rotations are used to rebalance tree.

AVL Rotations :-

We perform rotations in AVL tree only in case if Balance Factor is other than -1, 0 and 1.

There are Basically four types of rotations which are as follows :

1). L-L rotation :- Inserted node is in the left subtree of left subtree of A.

2). R-R rotation :- Inserted node is in the right subtree of right subtree of A.

3). L-R rotation :- Inserted node is in right subtree of left subtree of A.

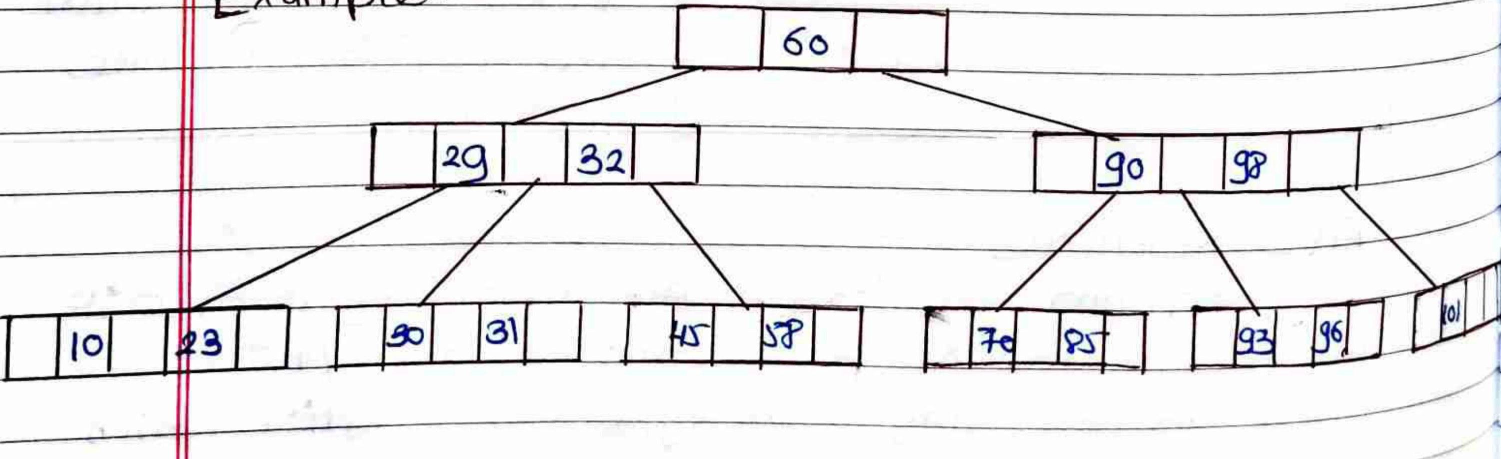
4). R-L Rotation :- Inserted node is in the left subtree of right subtree of A.

5). B Tree :- B Tree is specialized m-way tree that can be widely used for disk access. A B-tree of order m can have at most m-1 keys and m children.

Proporties :-

1. Every node in B-Tree contains at most m children
2. Every node in B-Tree except root node and leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

Example :-



Operations :-

1). searching :- The searching in B tree is similar to searching in Binary tree. for example, we search for an item 49 in following B Tree. The process will be:

①. compare item 49 with root node 78. since 49 < 78 hence, move its left sub-tree.

②. since, 40 < 49 < 56, traverse right subtree of 40.

③. 49 > 45, move to right compare 49.

④. match found, return.

Searching in B tree depends upon height of the tree. The search algorithm takes O(log n) time to search any element in B tree.

2). Inserting :- Insertion are done at leaf node level. The following algorithm needs to be followed in order to insert an item into B tree.

①. Traverse B tree in order to find appropriate leaf node at which node can be inserted.

②. If leaf node contains less than m-1 keys then insert element in increasing order.

③. Else, if leaf node contains m-1 keys, then follow following steps:

- Insert new element in increasing order of elements.
- split node into two nodes at median.
- Push median element upto its parent node.
- If parent node also contain m-1 number of keys, then split it too by steps.

Application of B tree :-

- B tree is used to index data and provides fast access to actual data stored on disks since, the to value stored in a large database that is stored on a disk is a very time consuming process.

searching an un-indexed and unsorted database containing n key values needs O(n) running time.

6). B + Tree :-

B+tree is an extension of B tree which allows efficient insertion, deletion and search operations.

The leaf nodes of B+ tree are linked together in form of the singly linked list to make search queries more efficient.

Advantages of B+ tree :-

1). Records can be fetched in equal number of disk accesses.

2) Height of tree remains balanced and less as compare to B tree.

3) we can access data stored in B+ tree sequentially as well as directly.

4) keys are used for indexing.

Graph :-

A graph can be defined as group of vertices and edges that are used to connect these vertices.

Defination :-

A graph G can be defined as an ordered set G(V, E) where V(G) represents set of vertices and E(G) represents set of edges.

which are used to connect these vertices.

Directed and Undirected Graph :-

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with directions with them.

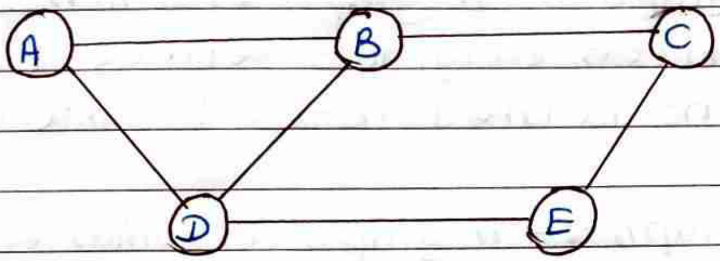


Fig : Undirected graph

As above figure edges are not attached with any of the directions.

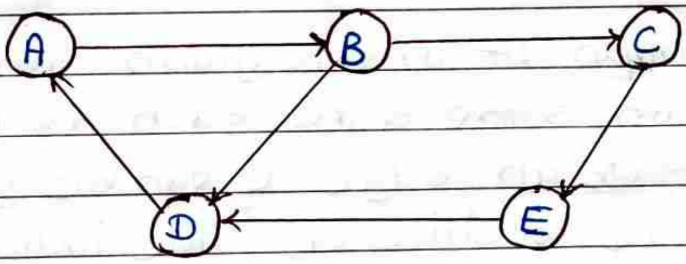


Fig: directed graph.

In above figure, directed graph edges form an ordered pair.

Graph Terminology :-

1). Path :- A path can be defined as sequence of nodes that are followed in order to reach some terminal node v from initial node u.

2). Closed path :- A path will be called as closed if initial node is same as terminal node. $v_0 = v_N$

3). Simple path :- If all nodes of graph are distinct with an exception $V_0 = V_N$, then such path P is called as closed simple path.

4). Cycle :- A cycle is a path which has no repeated edges or vertices except first and last vertices.

5). Connected graph :- A graph in which some path exists between every two vertices (u,v) in V. There are no isolated nodes in connected graph.

6). Complete graph :- A graph in which every node is connected with all other nodes. A complete graph contain $\frac{n(n-1)}{2}$ edges where n is number of nodes in graph.

7). Weighted graph :- In this graph each node is assigned with some data such as length or width The weight of an edge e can be given as w(e) which must be positive (+) value indicating cost of traversing edge.

8). Diagraph :- A diagraph is directed graph in which each edge is associated with some direction and traversing can be done only in specified direction.

9). Loop :- An edge that is associated with the similar end points can be called as loop.

10). Adjacent Nodes :- If two nodes u and v are connected via an edge e, then nodes u and v are called as neighbours or adjacent nodes.

11). Degree of a Node :- A degree of a node is a number of edges that are connected with that node. A node with degree 0 is called isolated.

Graph Representation :-

We simply mean, technique which is to be used to in order to store some graph into the computers memory.

1. Sequential Representation :- In this we use adjancy matrix to store mapping represented by vertices and edges. A graph having n vertices, will have a dimension n x n.

An entry M_{ij} in adjacency matrix representation of an ~~mat~~ undirected graph G will be 1 if there exists an edge between V_i and V_j.

An undirected graph and its adjacency matrix representation is shown in following :

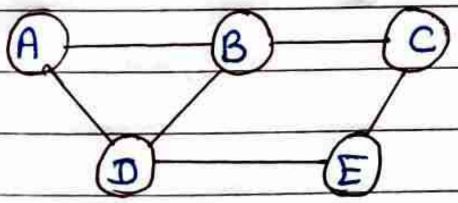


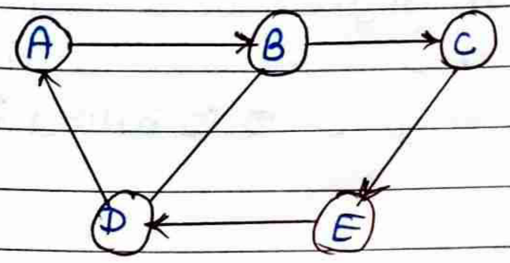
fig : Undirected graph

$$\begin{array}{c|ccccc|}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 0 & 1 & 0 \\
B & 1 & 0 & 1 & 1 & 0 \\
C & 0 & 1 & 0 & 0 & 1 \\
D & 1 & 1 & 0 & 0 & 1 \\
E & 0 & 0 & 1 & 1 & 0 \\
\hline
\end{array}$$

Fig: Adjacency matrix

In above figure, we can see mapping among vertices (A, B, C, D, E) is represented by using adjacency matrix which is also shown in fig.

A directed graph and its adjacency matrix representation is shown in figure :

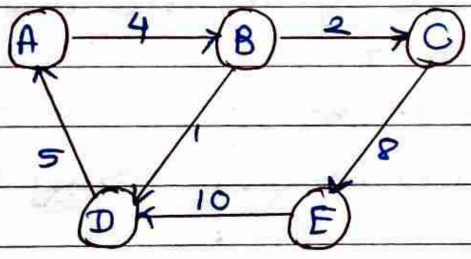


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

Fig : Directed Graph

Fig : Adjacency matrix

Representation of weighted directed graph is different Instead of filling entry by 1, non zero entries of adjacency matrix are represented by weight of respective edges.

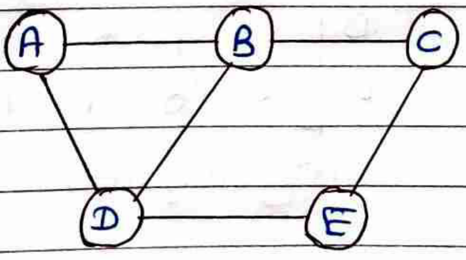


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

Fig : weighted directed graph

fig : Adjancy matrix

② linked representation :—



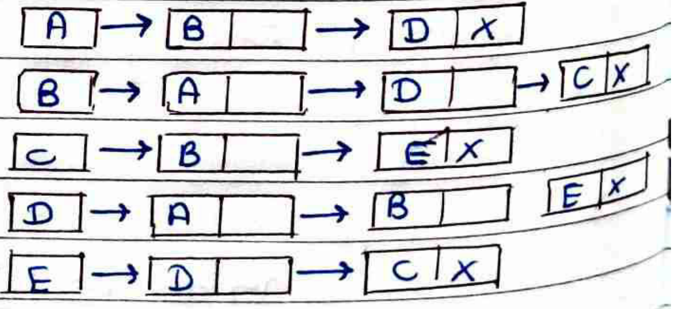
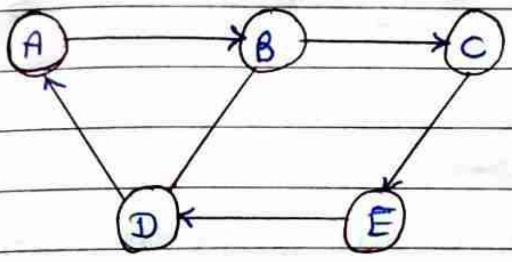


Fig : undirected graph

Fig : Adjacency list.

An adjacency list is maintained for each node present in graph which stores node value and a pointer to next adjacent node to respective node.



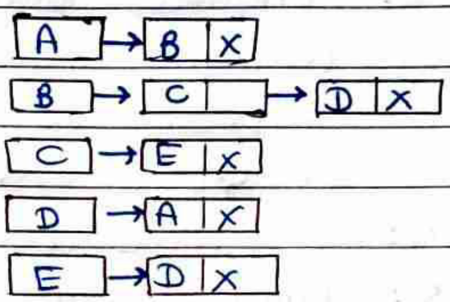


Fig : Directed graph   Fig : Adjjancy list

In directed graph, sum of lengths of all the adjjancy lists is equal to the number of edges present in the graph.

Graph Traversal Algorithm :-

In this tutorial we will learn all techniques by using which, we can traverse all the vertices of the graph. Traversing means examining all nodes and vertices of graph. There are two standard methods by using which, we can traverse graphs.

- Breadth first search
- Depth first search

①. Breadth first search (BFS) algorithm :-

Breadth first search is a graph traversal algorithm that starts traversing graph from root node and explores all the neighbouring nodes.

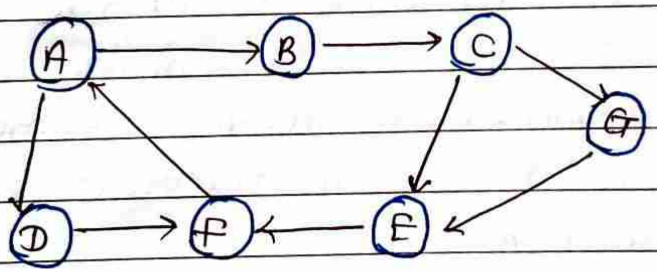
Then, it selects nearest node and explore all unexplored nodes. The algorithm follows same process for each of nearest node until it finds goal.

Algorithm :-

```
step 1:  SET STATUS = 1 (ready state)
         for each node in G.
step 2:  Enqueue starting node A & set its STATUS = 2
         (waiting state).
step 3:  Repeat steps 4 and 5 until
         QUE E is empty.
step 4:  Dequeue a node N. Process it & set its STATUS = 3
step 5:  Enqueue all neighbours of N that are in ready
         state (whose STATUS = 1) & set (STATUS = 2)
         [END OF LOOP].
step 6:  EXIT.
```

Example :-

consider graph G shown in following image, calculate minimum path P from node A to node E. Given that each edge has a length of 1.



Adjacency lists :

A	:	B, D
B	:	C, F
C	:	E, G
G	:	E
E	:	B, F
F	:	A
D	:	F.

solution :-

minimum Path P can be found by applying Breadth First search algorithm that will begin at node A and will end at E.

A → B → C → E

Depth First search Algorithm :-

DFS algorithm starts with initial node of graph G, & goes to deeper & deeper until we find goal node / node which has no children.

The data structure used in DFS is stack.

Algorithm :-

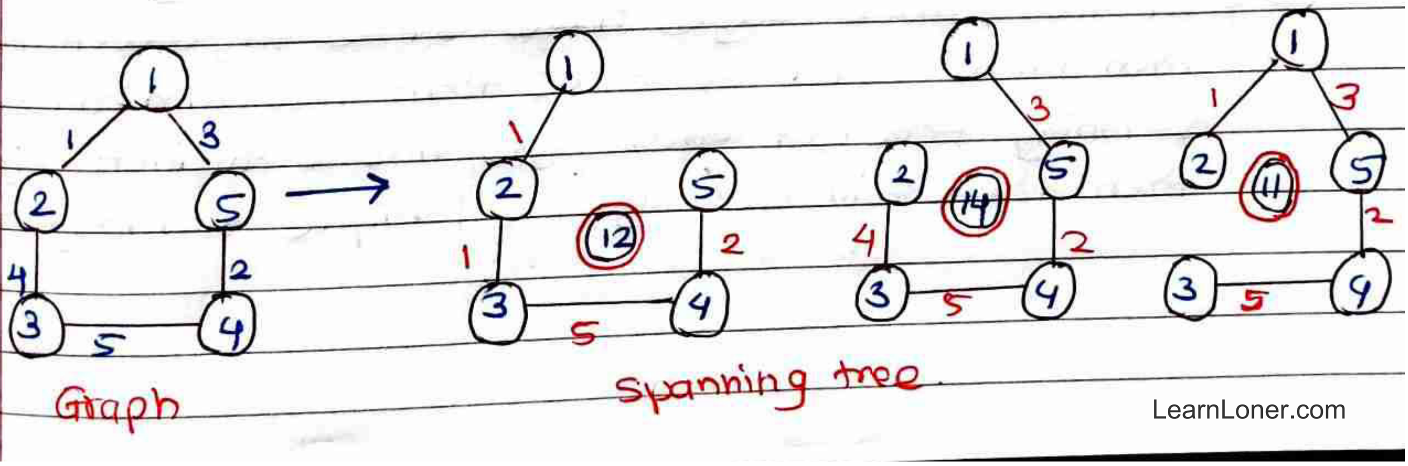
```
step 1: SET STATUS = 1 (ready state) for each node in G.
step 2: Push starting node A on stack & set its STATUS = 2
        (waiting state).
step 3: Repeat steps 4 and 5 until stack is empty.
step 4: Pop top node N. Process it & set its STATUS = 3.
step 5: push on stack all neighbours of N that are in
        ready state (whose STATUS = 1) and set their
        STATUS = 2 (waiting state) [END OF LOOP].
step 6: EXIT.
```

Spanning Tree :-

If we have a graph containing V vertices and E edges, then graph can be represented as: G(V, E). If we create spanning tree from above graph, then spanning tree would have same number of vertices as the graph, but vertices are not equal.

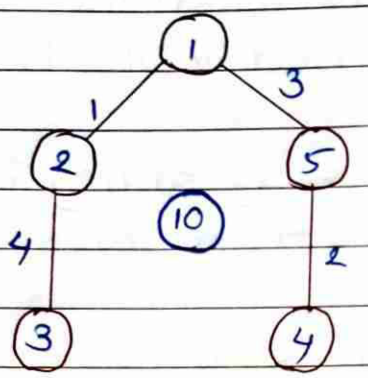
edges (spanning tree) = no. of edge (in graph) - 1.

Example :-



Minimum Spanning Trees :-

The minimum spanning tree is a tree whose sum of edge weights is minimum.



In above tree, total edge weight is less than above spanning trees, therefore a minimum spanning tree is a tree which is having an edge weight i.e 10.

Properties of spanning tree :-

- A connected graph can contain more than one spanining tree.
- All possible spanning trees that can be created from given graph G would have same number of vertices in given graph minus 1.
- Spanning tree does not contain any cycle. let's understand this property through an example.
- Spanning tree cannot be disconnected. If we remove one more edge from any of above spanning trees as.
- If two / more edges have same edge weight, then there would be more than two minimum spanning tree. If each edge has a distinct weight, then there will be only one / unique spanning tree

Applications of spanning tree :-

1). Building a network :- suppose there are many routers in network connected to each other, so there might be a possibility that it forms a loop.

2). clustering :- clustering means that grouping set of objects in such way that similar objects belong to same group than to different group. our goal is to divide the n objects into k groups such that distance between different groups gets maximised.

Searching :-

searching is a process of finding some particular element in list. If the element is present in the list, then process is called successful and process returns location of that element, otherwise search is called unsuccessful.

There are two methods widely used as below:

- Linear search
- Binary search

1). linear search :-

linear search is a simplest sequential search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of list with item whose location is to be found.

linear search is mostly used to search an unordered list in which items are not sorted. The algorithm is given as follows :

Algorithm :-

```
  LINEAR - SEARCH (A,N, VAL).
step 1 :- [INITIALIZE] SET POS = -1
step 2 :- [INITIALIZE] SET I =1
step 3 :- Repeat step 4 while I < =N
step 4 :- IF A[I] = VAL
            SET POS =1
            PRINT POS
            Go to step 6
            [END OF IF]
            SET I = I+1
            [END OF LOOP].
step 5 :- IF POS = -1
            PRINT "VALUE IS NOT PRESENT in ARRAY".
            [END OF IF].
step 6 :- EXIT.
```

Complexity of an Algorithm :-

Complexity	Best case	Average case	worst case
Time	O(1)	O(n)	O(n)
space			O(1)

C program of linear search :-

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

```
int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
int item, i, flag;
printf("\n Enter Item which is to be searched \n");
scanf("%d", item);
for (i=0; i<10; i++)
  {
    if (a[i] = =item)
     {
      flag =i+1;
      break;
     }
   else
      flag=0;
 }
   if(flag!=0)
    {
    printf("\n item found at location %d \n", flag);
    }
    else
    {
      printf("\n Item not found \n");
    }
 }
```

Output :-
```
Enter item which is to be searched
20
Item not found
Enter Item which is to be searched
23
Item found at location 2.
```

2). Binary Search :-

Binary search is a search technique which works on efficienty on sorted lists. Hence in order to search an element into some list by using binary search technique, we must ensure that list is sorted.

Binary search follows divide and conquer approach in which, list is divided into two halves and item is compared with middle element of list.

Binary search Algorithm :-

```
BINARY_SEARCH (A, lower-bound, upper-bound, VAL)
step 1 :- [INITIALIZE] SET BEG = lower-bound
              END = upper-bound , pos = -1.

step 2 :- Repeat steps 3 and 4 while BEG <= END

step 3 :- SET MID = (BEG + END)/2

step 4 :- IF A[MID] = VAL
              SET POS = MID
              PRINT POS  Go to step 6
              ELSE IF A[MID] > VAL
              SET END = MID - 1
              ELSE SET BEG = MID + 1

step 5 :- IF POS = -1
              PRINT "VALUE IS NOT PRESENT IN ARRAY"

step 6 :- EXIT.
```

Complexity :-

Sr.No.	Performance	Complexity
1).	Worst case	O (log n)
2).	Best case	O(1)
3).	Average case	O(log n)
4).	Worst case space complexity	O(1)

Example: Let us consider an array arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}. find location of item 23 in the array.

→ In 1st step :-

BEG = 0

END = 8 ron

MID = 4

a [mid] = a [4] = 13 < 23, therefore;

In second step :-

Beg = mid+1 = 5

End = 8

mid = 13/2 = 6

a[mid] = a [6] = 20 < 23, therefore ;

In third step :-

beg = mid+1 = 7

End = 8

mid = 15/2 = 7

a[mid] = a [7].

a[7] = 23 = item ;
therefore, set location = mid ;
The location of item will be 7.

item to be searched : 23

step 1 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

step 2 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

step3 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

In step 1 : a[mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13)2 = 6

In step 2 :- a[mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15/2 = 7.

step 3 :- a[mid] = 23
23 = 23
loc = mid.

C program : Binary search

```
#include<stdio.h>
int binary search (int [], int, int, int);
void main ()
```

```
{
 int arr[10] = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
 int item, location = -1;
 printf(" Enter the item which you want to search");
 scanf(" %d ", &item);
 location = binarySearch (arr, 0, 9, item);
 if (location ! = -1)
 {
  printf("Item found at location %d", location);
  }
 else
  {
   printf("item not found");
  }
 }
int Binary search (int a[], int beg, int end, int item)
{
 int mid;
 if (end >= beg)
  {
   mid = (beg + end)/2;
   if (a[mid] == item)
    {
     return mid+1;
    }
   else if (a[mid] < item)
   {
   return binary search (a, mid+1, end, item);
   }
  else
  {
   return binary search (a, beg, mid-1, item);
  }
 }
 return -1;
 }
```

Output:

```
Enter item which you want to search
19
Item found at location 2.
```

Sorting Algorithm :-

1) Bubble sort Algorithm :-

Bubble sort algorithm is a simplest sorting algorithm. Bubble sort works on repeatedly swapping of adjacent elements until they are not in intended order. It is called as Bubble sort because moment of array elements is just like movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly the array elements in bubble sort move to end in each iteration.

It is not suitable for large data sets. The average and worst case complexity of bubble sort is $O(n^2)$, where n is number of items.

Bubble sort is majorly used where :

- complexity does not matter
- simple and shortcode is preferred.

Algorithm :-

```
begin Bubblesort (arr)
  for all array elements
    if arr[i] > arr(i+1]
      swap(arr[i], arr[i+1]]
    end if
  end for
  return arr
end Bubble sort.
```

Bubble sort complexity :-

case	Time complexity	space complexity
Best case	o(n)	o(1).
Average case	o(n^2)	
worst case	o(n^2)	

Implementation of Bubble Sort :-

C language Implementation :-

```
#include <stdio.h>
void print(int a[],int n)
 {
  int i;
  for(i=0;i<n;i++)
   {
    printf("%d",a[i]);
   }
}
void bubble (int a[], int n)
 {
 int i,j,temp;
 for(i=0; i<n; i++)
  {
   For (j=i+1 ; j<n ; j++)
   {
   if (a [j] < a[i])
     {
     temp = a [i];
     a[i] = a[j];
     a[j] = temp;
     }
  }
 }
}
void main ()
 {
   int i,j,temp;
   int a[5] = { 10, 35, 32, 13, 26 };
   int n = size of (a) / size of (a0);
   printf ("Before sorting array elements are :\n");
   print (a,n);
   bubble (a,n);
   printf ("\n after sorting array elements-\n");
   print (a,n);
}
```

output :-

```
Before sorting array elements are -
10  35  32  13  26
After sorting array elements are -
10  13  26  32  35.
```

Bucket Sort Algorithm :-

The data items in the bucket sort are distributed in form of buckets.

Bucket sort is a sorting algorithm that seprates elements into multiple groups said to be buckets. Elements in bucket sort are first uniformly divided into groups called buckets, and then they are sorted by any other sorting algorithm. After that, elements are gathered in sorted manner.

Advantages of bucket sort are :-

- Bucket sort reduces no. of comparisons
- It is asymptotically fast because of uniform distribution of elements.

limitations of bucket sort are :

- It may or may not be a stable sorting algorithm
- It is not useful if we have a large array bcz it increases the cost.
- It is not an in-place sorting algorithm, because more some extra space is required to sort the buckets.

The best and average-case complexity of bucket sort is O(n+K), worst-case complexity of bucket sort is $O(n^2)$, where n is number of items.

Bucket sort is commonly used :

- with floating - point values.
- when input is distributed uniformly over a range.

Algorithm :-

```
Bucket sort (A [])
1.  let B [0 .... n-1] be a new array.
2.  n = length [A].
3.  for i=0 to n-1
4.  make B[i] an empty list
5.  for i=1 to n
6.  do insert A[i] into list B[n a[i]]
7.  for i=0 to n-1
8.  do sort list B[i] with insertion sort.
9.  concatenate lists B[0], B[1] .... B[n-1] together
    in order.
10. END.
```

Complexity :-

1. Time complexity :-

case	time complexity
Best case	O(n+k).

Average case	O(n+k)
worst case	$O(n^2)$

2. space complexity :-

Space complexity	O(n*k)
stable	YES

Implementation of bucket sort in c:-

```
#include <stdio.h>
int getMax (int a[],int n)
 {
  int max =a[0];
  for( int i=1;i<n;i++)
    if (a[i]>max)
       max= a[i];
   return max;
 }
void bucket(int a[], int n)
 {
  int max=getmax(a,n)
  int bucket[max],i;
  for( int i = 0 ;i<=max ;i++)
   {
        bucket [i]=0 ;
    }
   for(int i=0 ; i<n ; i++)
    {
     bucket [a[i]]++ ;
    }
   for (int i= 0 ; j=0 ; i<=max ;i++)
    {
    while (bucket [i] >0)
      {
      a[j++] = i ;
      bucket [i]-- ;
      }
   }
  }
  void printArr (int a[],int n)
   {
    for (int i=0; i<n ; i++)
    printf("%d", a[i]);
   }
  int main ()
  {
   int a[]= {54, 12, 84, 57, 69, 41, 9 ,5};
   int n= size of(a) / size of a[0]) ;
   printf ("Before sorting array elements are:-\n");
   printArr(a,n);
   bucket(a,n);
   printf( "\n After sorting array elements are:-\n");
    printArr(a,n);
   }
```

output :-

```
Before sorting array elements are :-
54  12  84  57  69  41  9  5
After sorting array elements are :-
5  9  12  41  54  57  69  84.
```

Heap Sort Algorithm :-

Heap sort processes the elements by creating min-heap or max-heap using the elements of the given array.

Heap sort basically recursively performs two main operations :

- Build a heap H, using the element of array.
- Repeatedly delete the root element of heap formed in 1st phase.

What is heap ?

A heap is a complete binary tree, and binary tree is a tree in which node can have utmost two children.

Algorithm :-

```
Heap Sort (arr)
BuildMaxHeap (arr)
for i = length (arr) to 2
    swap arr [i] with arr [1]
     heap-size [arr] = heap-size [arr] ? 1
      MaxHeapify (arr, 1)
End.
```

```
BuildMaxHeap(arr):
    BuildMaxHeap (arr)
        heap-size (arr)= length (arr)
        for i = length (arr) / 2 to 1.
    maxHeapify (arr, i)
    End.
```

complexity :-

case	Time complexity	space complexity
Best	o (n log n)	o(1).
Average	o (n log n)	
worst	o (n log n)	

Implementation of Heap sort :-

```
#include<stdio.h>
/* function to heapify a subtree. Here is 'i' the
index of root node in array a[], and 'n' is size
of heap */
void heapify (int a[], int n, int i)
 {
  int largest = i
  int left = 2*i+1
  int right = 2* i+2
  if (left <n && a[left]> a[largest]
    largest= left;
```

```
if (right<n && a[right]>a[largest]
  largest = right ;
if (largest != i)
{ int temp = a[i] ;
   a[i] = a [largest];
   a [largest] = temp;
   heapify (a, n, largest);
 }
}
void heapsort (int a [], int n)
{
  for(int i = n/2 -1 ; i>=0, i--)
    heapify (a, n, i);
  for (int i= n-1; i>0 , i--)
  { int temp = a [0];
     a[0] = a[i] ;
     a [i] = temp;
  heapify (a, i, 0);
 }
} void printArr (int arr [], int n).
{
   for(int i= 0; i<n ;++i)
     {
       printf("%d", arr[i]);
       printf(" ");
     }
} int main ()
 {
 int a[] = {48, 10, 23, 43, 28, 26, 1};
 int n = sizeof (a) / sizeof (a[0]);
 printf ("Before sorting array elements are - \n");
```

```
    printArr(a,n);
    heapSort(a,n);
    printf("\nAfter sorting array elements are -\n");
    printArr(a,n);
    return 0;
}
```

output:
```
Before sorting array elements are :-
 48 10 23 43 28 26 1
After sorting array elements are -
 1 10 23 26 28 43 48.
```

Insertion sort Algorithm :-

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card. The idea behind the insertion sort is that first make /take one element, iterate it through sorted array. complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is number of items.

Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort and merge sort etc.

Insertion sort has various advantages such as :

- simple implementation.
- Efficient for small data sets.
- Adaptive i.e it is appropriate for data sets that are already substantially sorted.

complexity :-

case time complexity :

Best case o(n)

space complexity o(1).

Avergge case $o(n^2)$

worst case $o(n^2)$.

Implementation of insertion sort :-

```
#include <stdio.h>
void insert (int a[], int n)
 {
   int i, j, temp;
   for (i= 1; i<n ,i++) {
     temp = a[i];
      j = i-1 ;
    while (j>=0 && temp <= a[j])
    {
     a[j+1] = a[j];
     j = j-1;
    }
    a[j+1] = temp;
  }
}
void printArr (int a[], int n)
 {
   int i;
   for(i=0; i<n; i++)
    printf ("%d", a[i]);
 }
int main()
{
  int a[] = {12,31, 25, 8, 32, 17};
```

```
    int n = size of (a) / size of a[0]);
    printf ("Before sorting array elements are -\n");
    printArr (a, n);
    insert (a, n);
    printf ("\nAfter sorting array elements are -\n");
    printArr (a, n);
    return 0;
}
```

output:

```
Before sorting array elements are -
12  31  25  8  32  17
After sorting array elements are -
8  12  17  25  31  32.
```

Merge sort Algorithm :-

Merge sort is the sorting technique that follows divide and conquer approach. This will be very helpful and interesting.

merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort elements.

Algorithm :-

arr is given array, beg is starting element and end is last element of array.

```
MERGE - SORT (arr, beg, end)
 if beg < end
 set mid = (beg + end)/2.
 MERGE - SORT (arr, beg, mid)
 MERGE_SORT (arr, mid+1, end)
 MERGE (arr, beg, mid, end)
```

end of if
End MERGE_SORT.

implementation of merge sort:-

```
/* function of merge the subarrays of a[] */
void merge (int a[], int beg, int mid, int end)
 {
   int i, j, k;
   int n1 = mid - beg + 1;
   int n2 = end - mid;
   int LeftArray [n1], RightArray [n2];
   /* copy data to temp arrays */
   for(int i=0; i< n1; i++)
   LeftArray [i] = a[beg + i];
   for (int j=0; j<n2; j++)
   RightArray[j] = a[mid+1 +j];
   i = 0;
   j = 0;
  k = beg;
 while (i< n1 && j<n2)
  {
   if(LeftArray [i]<= RightArray [j])
    {
     a[k] = LeftArray[i];
      i++;
    }
  else
   {
   a[k]=RightArray [j];
   j++;
   }
  }
   k++;
  }
 while (i<n1)
 {
  a[k]= LeftArray[i];
   i++;
  k++;
 }
while (j<n2)
{
  a[k] = RightArray [j];
  j++;
  k++;
 }
}
```

Complexity :-

case	Time complexity	space complexity
Best case	o(n*logn)	o(n).
Avg. case	o(n*logn)	
worst case	o(n*logn)	

DATA STRUCTURE CODING QUE.

Arrays by using c :

1). program to demonstrate arrays in c

```
#include <windows.h>
# include < stdio.h>
# include < stdlib.h>
# define NUM-EMPLOYEE 10
int main (int argc, char *arg v []}
{ int salary [NUM-EMPLOYEE], lcount=0,
    gcount=0 , i=0;
   printf ("Enter employee salary (MAX 10) \n");
      for (i=0; i< NUM EMPLOYEE ; i++)
  {
    printf ("\n Enter employee salary : %d -",
            i+1) ;
    scanf ("%d", & salary [i]);
  }
      for (i=0; i<NUM-EMPLOYEE ; i++)
  {   if (salary [i] <3000]
      lcount ++ ;
    else
      gcount ++;
  }
   printf ("\n There are {%d} employee with
            salary more than 3000 \n", g count);
  printf ("There are {%d} employee with salary
            less than 3000 \n", lcount);
  printf ("press Enter to continue .. \n");
```

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

27. Linked list in c++ :

```
  using namespace std;
   template <typename T>
   class node
   {
     public :
     T value ;
     Node *next;
     Node *previous ;
     Node (T value)
     {
       this -> value = value ;
     }
  };
   template < typename T >
   class linked list
   {
     private :
     int size ;
     Node <T> *head_ = NULL ;
     Node <T> *tail_= NULL;
     Node <T> *itr = NULL ;
     public ;
     Linked list ()
     {
      this -> size_ = 0;
     }
```

```
      void append (T value)
       {
        if (this->head == NULL)
       {
        this->head_ = new Node<T> (value);
        this->tail = this->head_;
        }
      else
       {
      this->tail_->next = new node<T>(value);
      this->tail_->next->previous = this_
                                  ->tail;
       }
      this->size_+ =1;
      }
     void prepend (T value)
      {
     void resetIterator ()
     {
     tail_ = NULL;
     }
int main (int argc, char **argv)
 {
      LinkedList <int> lList;
       llist.append (10);
      llist.append (3);
       llist.append (1);
    cout<< "printing linked list <<endl;
    cout<<endl;
    return 0;
  }
```

3). stack implementation in c :

```
#include <stdio.h>
int MAXSIZE = 8 ;
int stack [8] ;
int top = -1 ;
int isempty () {
    if (top == -1)
       return 1 ;
    else
       return 0 ;
    }
 int isfull () {
     if (top == MAXSIZE)
        return 1 ;
     else
        return 0 ; }
 int peek () {
     return stack [ top] ; }
 int pop () {
     int data ;
     if(!isempty()) {
      data = stack [ top] ;
      top = top-1 ;
      return data ; }
     else {
      printf ( "could not retriere data, stack is empty \n");
     }
}
int push (int data) {
   if (!isfull()) {
   top = top+1 ;
```

```
    stack[top] = data ;
    } else {
       printf("could not insert data, stack is full \n");
    }
}
int main () {
    //push items on to the stack
    push (3) ;
    push (5) ;
    push (9) ;
    push (1) ;
    push (12) ;
    push (15) ;
    printf("Element at top of the stack : %d \n", peek()
    printf(" Elements : \n");
   // print stack data
   while (!isempty ()) {
       int data = pop ();
        printf("%d /n ", data) ;
   }
  printf("stack full : %s \n", isfull () ? "true" : "false");
  printf("stack empty : %s \n" ,isempty() "true", "false
  return 0;
}
```

DATA STRUCTURES INTERVIEW QUESTIONS WITH ANSWERS

Q.1. What is data structure ?

→ A data structure is a way of organizing data that considers not only items stored, but also their relationoship to each other.

Q.2. List out the areas in which data structure are applied extensively ?

→ • compiler design , • operating system,
• database system , • statistical analysis,
• numerical analysis , • artificial intelligence

Q.3. What are major data structures used in following areas Rdbms, network data model and Hierarchical data model.

→ Rdbms = array (array of structures).
network data model = graph.
Hierarchical data model = tree.

Q.4. If you are using c language to implement the heterogeneous linked list, what pointer type will you use ?

→ The heterogeneous linked list contains different data types in its nodes and we need a link, pointer to connect them. It is not possible to use ordinary pointer for this. so we go for void pointer. Void pointer is capable of storing pointer to any type as it is a generic pointer type.

Q.5. minimum number of queues needed to implement the priority queue?

→ two. one queue is used for actual storing of data and another for storing priorities.

Q.6. What is data structure used to perform recursion?

→ stack. because of its LIFO (Last IN first Out) property it remembers its 'caller'

Q.7. What are notations used in evaluation of arithmatic expressions using prefix & postfix forms?

→ Polish and Reverse polish notations.

Q.8. convert expression ((a+b)*c-(d-e)^(f+g)) to equivalent prefie and postfix notations.

→ prefix notation : - * + abc ^ - de + fg

postfix notation : ab+c*de - fg + ^ -

Q.9. what are methods available in storing sequential files?

→ 1. straight merging,

2. natural merging,

3. polyphase sort,

4. distribution of initial runs.

Q.10. whether linked list is a linear or non-linear data structure?

→ According to access strategies linked list is a linear one. according to storage linked list is a non linear one.

Q.11. define doubly linked list.

→ It is collection of data elements called nodes, where each node is divided into three parts :

- an info field that contains information stored in the node.
- left field that contain pointer to node on left side.
- Right field that contain pointer to node on right side.
-

Q.12. What are the Issues that hampers efficiency in sorting a file ?

→
- length of time required by programmer in coding a particular sorting program.
- amount of machine time necessary for running the particular program.
- amount of space necessary for particular pgm.
- object oriented analysis and design.

Q.13. calculate efficiency of sequential search ?

→ The number of comparisons depends on where the record with argument key appears in table

- If it appears at first position then one comparison.
- If it appears at last position then n comparison.
- average = $\frac{n+1}{2}$. comparisons.

- number of comparisons in any case is O(n)

Q.14. Is any implicit arguments are passed to a function when it is called ?

→ yes, there is a set of implicit arguments that contain information necessary for function to execute and return correctly, one of them is return address which is stored within the function's data area, at time of returning to calling program address is retrived and function branches to that location.

Q.15. Paranthesis is never required in postfix or prefix expressions ? Why

→ parenthesis is not required because order of the operators in postfix / prefix expressions determines actual order of operations in evaluating expression.

Q.16. List out few of applications of tree data structure ?

→ The manipulation of arithmatic expression, symbol table construction & syntax analysis.

Q.17. List out few of applications that make use of multilinked structures ?

→ sparse matrix, Index generation.

Q.18. What is type of the algorithm used in solving 8 queens problem ?

→ backtracking.

Q.19. In an AVL Tree, at what condition balancing is to be done?

→ If 'pivotal value' or height factor is greater than 1 or less than -1.

Q.20 In Rdbms, what is the efficient data structure in internal storage representation.

→ b + tree. because b+ tree, all the data is stored in only in leaf nodes, that makes searching easier. this corresponds to records that shall be stored in leaf nodes.

Q.21. what is difference between array and a stack?

→ Stack follows LIFO. thus the item that is first entered would be last to be removed.

In the array, items can be entered or removed by in any order. basically, each member access is done using index. no strict order is to be followed here to remove a particular element.

Q.22. How to check whether a linked list is circular?

→ create two pointers, each set to start of list. update each as follows:

```
while (pointer 1)
{
 pointer 1 = pointer 1 -> next;
 pointer 2 = pointer 2 -> next;
 if (pointer 2) pointer 2 = pointer 2 -> next;
if (pointer 1 == pointer 2)
{
print("circularn");
}
}
```

Q.23. What is a node class?

→ A node class is class that, relies on the base for service and implementation, provides a wider interface to users than its base class, relies primarily on virtual functions in its public interface depends on all its direct and indirect base class.

Q.24. When can you tell that a memory leak will occur?

→ a memory leak occurs when a program loses the ability to free a block of dynamically allocated memory.

Q.25. What are types of collision Resolution techniques and methods used in each of the type?

→ open addressing (closed hashing), methods used include: overflow block. closed addressing (open hashing) methods used include: linked list, binary tree

Q.26. Which is simplest file structure? (sequential, index, random).

→ Sequential is the simplest file structure.

Q.27. What are the notations, used in evaluation of arithmatic expression, using prefix and postfix forms?

→ Polish and Reverse polish notations.

Q.28. list out few of applications of tree data structure?

→ The manipulation of arithmatic expressions, symbol table construction and syntax analysis.

Q.29. difference between calloc and malloc?

→ malloc: allocate n bytes.
calloc: allocate m times n bytes initialized to 0.

Q.30. which file contains the definition of member functior

→ defination of member function for the linked list class are contained in linkedlist.cpp file.

Q.31. How is the front of the queue calculated?

→ The front of the queue is calulated by
front = (front +1) % size.

Q.32. Why is the Isempty() member method called?

→ the isempty() member method is called within the dequeue process to determine if there is an item in dequeue to be removed i.e. isempty() is called to decide whether queue has atleast one element. This method is called by dequeue() method before returning front element.

Q.33. Which process places data at back of queue?

→ enque is a process that places data at back of the queue.

Q.34. What is queue ?

→ A queue is sequential organization of data. a queue is a first in first out type of data structure. an element is inserted at last position and an element is always taken out from first position.

Q.35. What does isempty () member method determine

→ isempty () checks if stack has at least one element. this method is called by pop () before retrieving and returning top element.

Q.36. What method removes value from top of a stack ?

→ The pop() member method removes value from top of a stack, which is then returned by the pop() member method to statement that calls pop() member method.

Q.37. What method is used to place a value onto the top of a stack ?

→ push() method, push is the direction that data is being added to stack. push () member method places a value onto the top of a stack.

Q.38. How do you assign an address to an element of a pointer array ?

→ We can assign a memory address to an element of a pointer array by using the address operator, which is ampersand (&), in an assignment statement such as ptemployee [0] = &projects [2];

Q.39. How many parts are there in a declaration statement ?

→ There are two main parts, variable, identifier & data type and third type is optional which is type qualifier like signed / unsigned.

Q.40. list some of the static data structures in c ?

→ some of the static data structures in c are arrays, pointers, structures etc.

Q.41. define dynamic data structure ?

→ A data structure formed when number of data items are not known in advance is known as dynamic data structure or variable size data structure.

Q.42. list some of dynamic data structures in c ?

→ some of dynamic data structures in c are linked lists, stack, queues, trees etc.

Q.43. define linear data structure.

→ linear data structures are data structures having a linear relationship between its adjacent elements.

eg: linked list.

Q.44. define non-linear data structures.

→ Non linear data structure are the data Structures are data structure that don't have a linear relationship between its adjacent elements but have a hierarchical relationship between the elements.

eg : trees and graphs.

Q.45. state the different types of linked lists?

→ The different types of linked list include singly linked list, doubly linked list and circular linked list.

Q.46. List the basic operations carried out in a linked list?

→
- creation of a list
- Insertion of a list.
- deletion of a node.
- modification of a node.
- traversal of a node.

Q.47. define a stack.

→ Stack is an ordered collection of an elements in which insertion and deletions are restricted to one end. The end from which elements are added and or removed is referred as top of stack.

Q.48. List out the basic operations that can be performed on a stack.

→
- push operation.

- pop operation
- peek operation
- empty check
- fully occupied check.

Q.49. State the different ways of representing expression

→
- Infix notation.
- prefix notation
- postfix notation.

Q.50. what is sequential search ?

→ In sequential search each item in the array is compared with the item being searched until a match occurs.

THANK YOU !!!

keep following us on our

Social Media Channels

for More Helpful Contents !

😊