A Beginner's Guide to

DATA STRUCTURES AND ALGORITHM

Prepared By

Divyanshu Dobhal

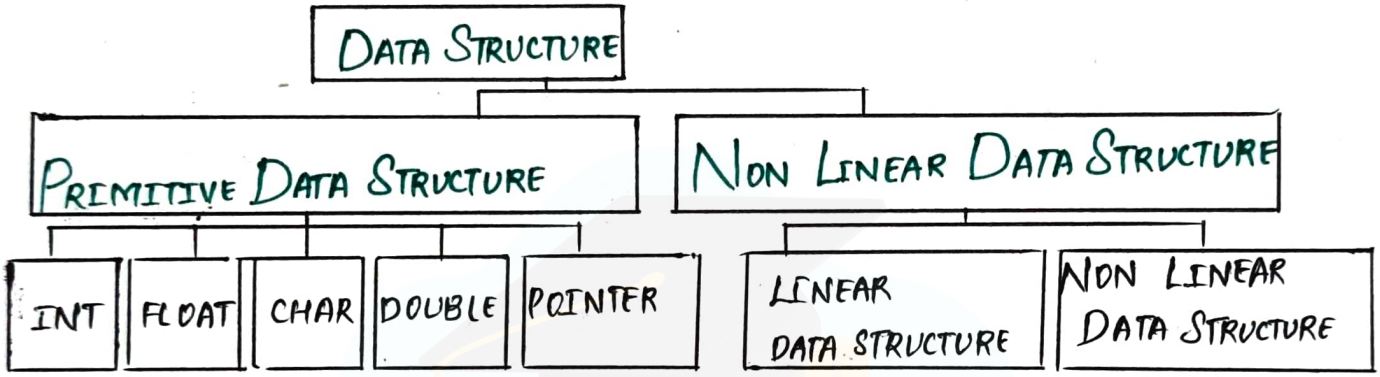
INDEX

What is a DataStructure

DataStructure is a way of storing and organising data. so that it can be used efficiently

As per the name indicates itself organise the data in memory

It is a set of algorithms that we can use in any programming language to structure data



CLASSIFICATION OF DATASTRUCTURE

LINEAR DATA STRUCTURE

The arrangement of data in sequential manner is known as Linear DataStructure

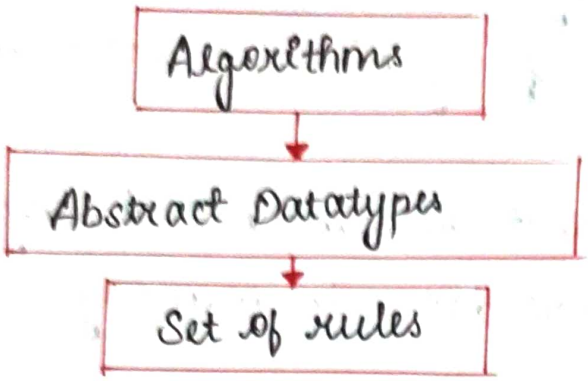
Examples: Arrays, Stacks, Queue, LinkedList

NON LINEAR DATA STRUCTURE:

When one element is connected to 'n' number of elements it is known as Non Linear Datastructure

Examples: Trees, graphs

ALGORITHMS AND ABSTRACT DATA TYPES



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

Abstract Data Types tells what is to be done
Data Structure tells how is to be done

ADT gives us a blue print while Data structure provides Implementation part

What is Data?

Data is defined as Elementary value / collection of values

Example:

Students name and students Id

What is a Record?

Record is defined as collection of various data items

Example:

Student name, entity, address, course and marks can be grouped to geather to form Record

What is a File?

File is a collection of various records of one type of entity

Example:

If there are 60 employees in a class then there will be 20 records related file where record contains info of employees

What is attribute and Entity?

- Entity represents class of certain object
- Entity contains various attributes
- Each attribute represents property of entity

What is the need of data Structure

As applicants are getting complexed and amount of data is increasing day by day

The following problem may arise:

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

Data Structures: consider an inventory size of 100 items in a store. if our application needs to transverse 100 items every time, results in slowing down process

Multiple Requests: If thousands of users are searching data simultaneously on a web server, during that process.

To solve these problems, Data Structure is used.

Data is organised to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly

Advantages of Data Structure

- Efficiency
- Reusability
- Abstraction

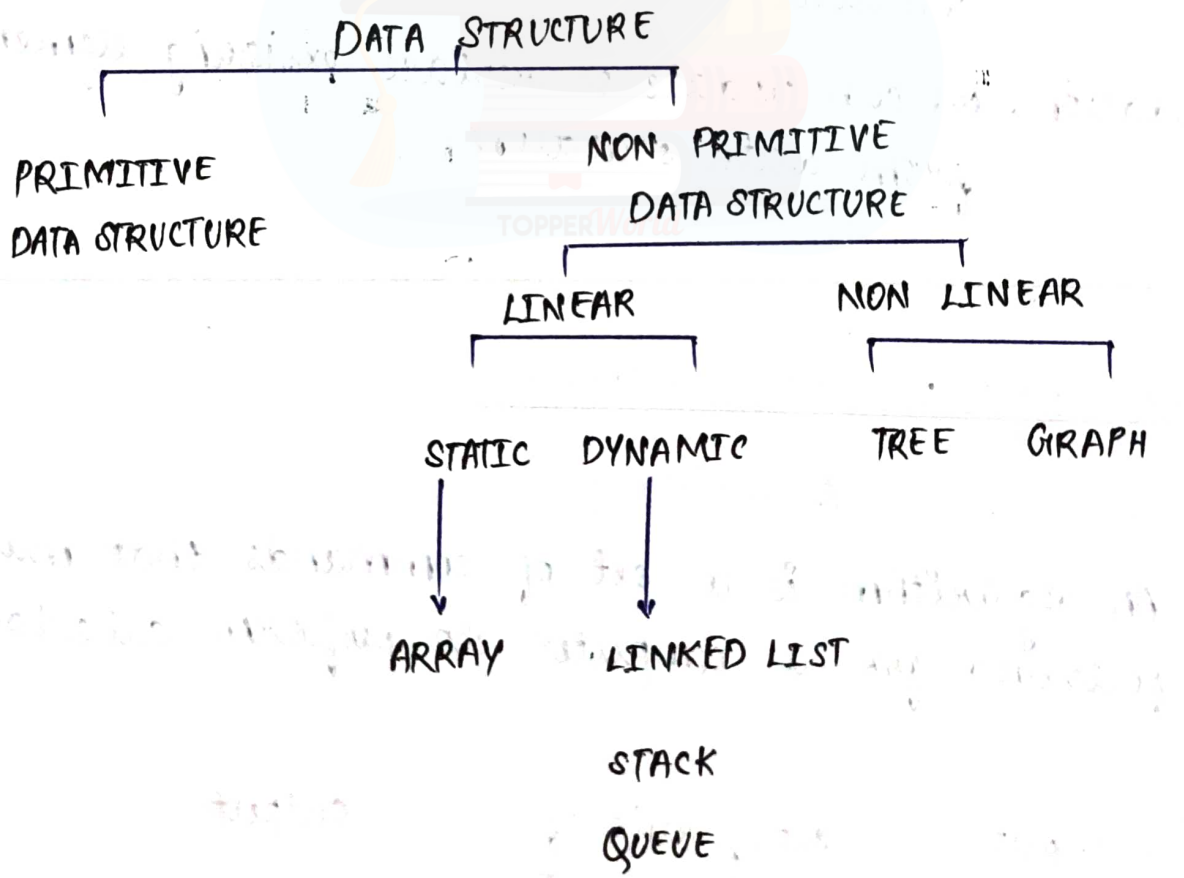
Efficiency - Data Structure is a secure way of storing data on our system

It helps us process data easily

Reusable - Data Structures are reusable ie once we have implemented particular data structure, we can use it at any place.

Abstraction - It allows users to work with data structures without having to implementation details, which simplify programming

DATA STRUCTURE CLASSIFICATION



Operations On Data Structure:

Traversing - we can access an element of data structure at least once

Searching - we can easily search for data element in datastructure

Sorting - we can sort elements either in ascending or descending order

Insertion - we can insert new data elements in data structure

Deletion - we can delete data elements from data structure

Updation - we can update or replace exsisting elements from data structure

Data Structure And Algorithm

What is an Algorithm?

An algorithm is a set of commands that must be followed for a computer to perform calculations

Input → set of rules of obtain output from given input → Output

CHARACTERISTICS OF AN ALGORITHM:

Input - Algorithm take input data, which can be in various formats, such as number, text

Processing - The algorithm processes the input data through series of logical and mathematical operations

Output - After processing is complete, algorithm produces output.

Efficiency - Aiming to accomplish tasks quickly

Optimization - optimize algorithm to make them fast and reliable.

Implementation - Algorithm are implemented in various programming languages, enabling computers to execute them and produce outputs.

ALGORITHM APPROACHES

BRUTE FORCE ALGORITHM

DIVIDE AND CONQUER

GREEDY ALGORITHM

ALGORITHM ANALYSIS

Algorithm analysis involves studing the efficiency and performance of algorithms in terms of their time complexity and space complexity

By analyzing algorithms we can determine how they scale with input size and resources, helping us choose the most suitable algorithm for a specific problem.

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.

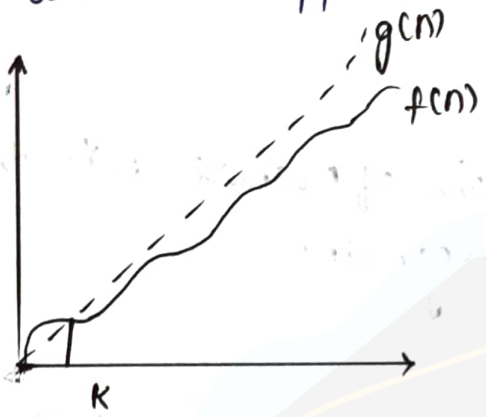
Average case - It takes average time for the program execution

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

ASYMPTOTIC NOTATIONS: used for calculating runtime complexity of an algorithm

① Big oh notation (O): This measures the performance of an algorithm by simply providing the order of growth of 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 function defined for positive integer.

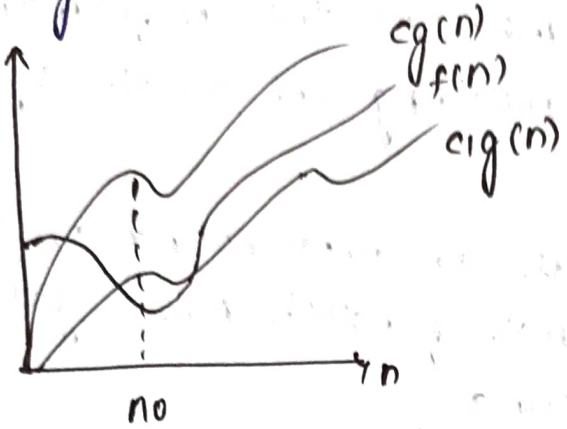
then $f(n) = O(g(n))$ as $f(n)$ is big oh of $g(n)$ or $g(n)$ is an order of $g(n)$ if there exsists constants c and no such that

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

② Omega Notation (Ω):

It is basically describes best case scenario which is opposite to big o notation. It is time formal

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 the steps required to execute programs

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

The above condition is satisfied only if when

$c_1 \cdot g(n) <= f(n) <= c_2 \cdot g(n)$

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 point the address of the value stored anywhere in the computers memory

Pointer arithmetic

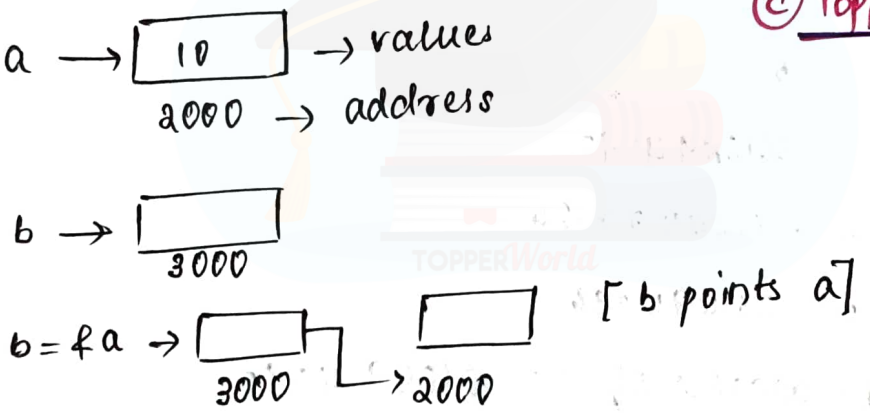
Four arithmatic operators can be used in pointers: ++, --, +, -

Array of pointers:

you can define array of hold number of pointers

pointer to pointer:

c allows you to have pointer on a pointer and so on.



POINTER PROGRAM

```
#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("value 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
value of a = 3010494292
address of a = -1284473004
address of b = 3014994296
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("value 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 = 2831585120

address of c = 2831685128

STRUCTURE

A structure is a composite datatype that defines a grouped list of variables that are placed under one name in block of memory

PROGRAM STRUCTURE

```
struct structure_name
{
 data_type member1;
 data_type member2;
   .
   .
 data_type member;
 };
```

ADVANTAGES OF STRUCTURE:

It can hold variables of different datatype

We can create objects containing different types

It allows to re-use the datalayout 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);
scanf("%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

ARRAYS:

Arrays are defined as collection of similar datatypes Array is the simplest data structure where each element can be randomly accessed by using its index number

ARRAY DECLARATION:

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

PROGRAM WITHOUT USING ARRAY:

```
#include<stdio.h>
 void main()
 {
  int marks;
  marks1= 56;
int marks_2 =78; marks_3 = 89;
  float avg =(marks_1+ marks_2 + marks_3)/3;
  print (avg);
  }
```

PROGRAM WITH USING ARRAY

```
#include<stdio.h>
 void main(){
   int marks[3] = {56,57, 89};
   int i;
   float avg;
```

```
for( i=0; i<3; i++)
{
avg = avg + marks[i];
}
printf(avg);
}
```

COMPLEXITY OF ARRAY OPERATION

① Time Complexity

Algorithm	Average 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

The array space complexity for worst case is o(n).

MEMORY ALLOCATION OF ARRAY

Each element in array represented by indexing

Indexing of array can be defined in three ways.

1. 0(zero Based Indexing)
   The first element of 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 random index

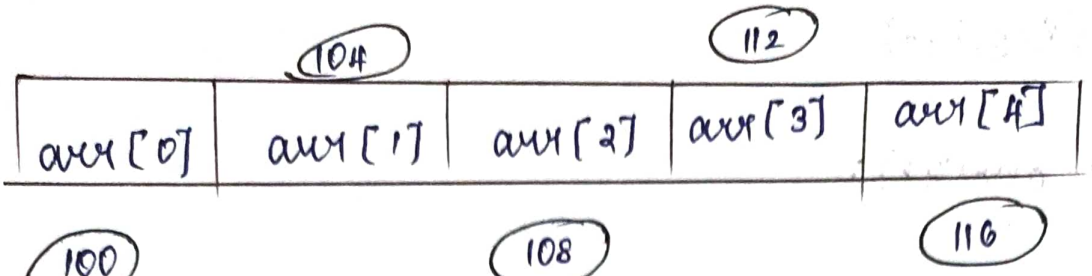


Figure: int arr[5]

ACCESSING ELEMENTS OF ARRAY

To access any random element of an array we need to follow 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 1D array can be calculated

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

Example:

In an array, A[-10...2] Base address (BA) = 888, size of an element = 2 bytes, find location of A[-1]

$$L(A[-1]) = 888 + [(-1) - (-10)] \times 2$$

$$= 888 + 18$$

$$\therefore \text{location of } A[-1] = 1017$$

2D ARRAY:

2D array can be defined as an array of arrays The 2D array is organised as an array of arrays.

How to declare 2D array

The syntax for declaration of two dimension array is

```
int arr[max_rows][max_columns]
```

However it produces data structures which looks like

	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]

Figure : a[n][n]

HOW TO ACCESS DATA IN 2D ARRAY

Due to the fact that elements in 2D array can be randomly accessed

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

where i, j are rows and columns

PASSING ARRAY TO FUNCTION:

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

PROGRAM

```
#include<stdio.h>
void 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;
 }
```

INITIALISING 2D ARRAY:

The syntax to declare and initialize 2D array

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

number of elements in 2D array = no of rows * no of columns

MAPPING 2D ARRAY TO 1D ARRAY

The size of two dimensional array is equal to multiplication of number of rows and number of columns presents in array

3x3 two dimensional array is shown.

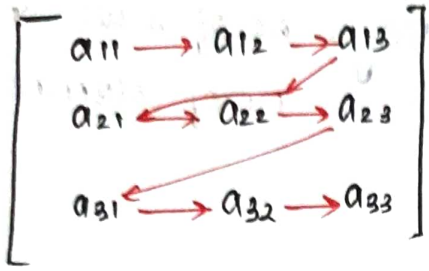
	0	1	2 ← COLUMN INDEX
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

↑ ROW INDEX

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

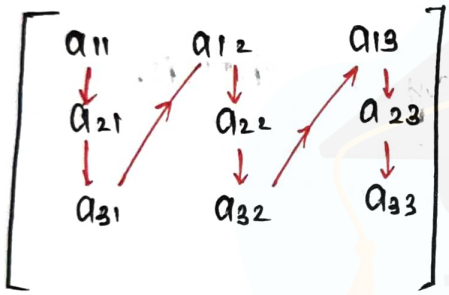
Row major ordering:

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



Column major ordering

According to column major ordering, all columns of 2D array are stored in memory



Calculating address of random elements of 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 column then address of element a[i][j] is calculated as

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

B.A is Base Address

2) By column major order:

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

LINKED LIST

why there is a need of linked list?

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

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

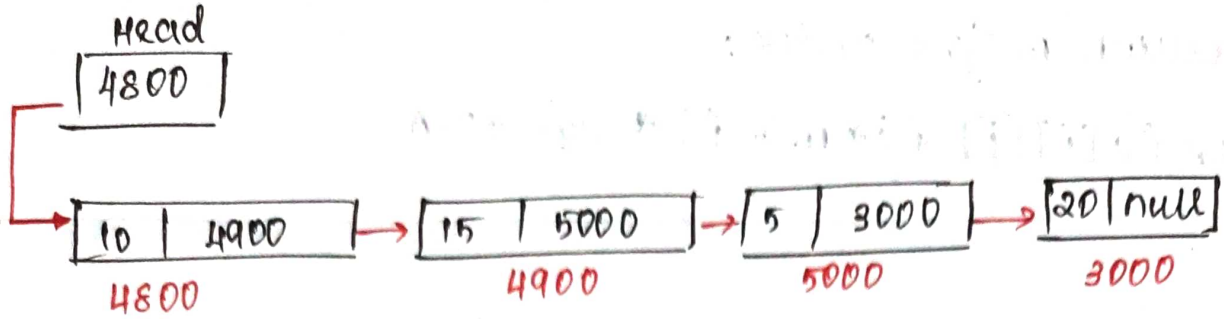
DRAWBACKS OF USING ARRAY

we cannot insert more than 3 elements in above example because only 3 spaces are allocated by 3 elements.

In arrays, wastage of memory occurs.

WHAT IS LINKED LIST?

A linked list is also a collection of elements but the elements are not stored in consecutive location. It has two nodes. one is data, other is 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

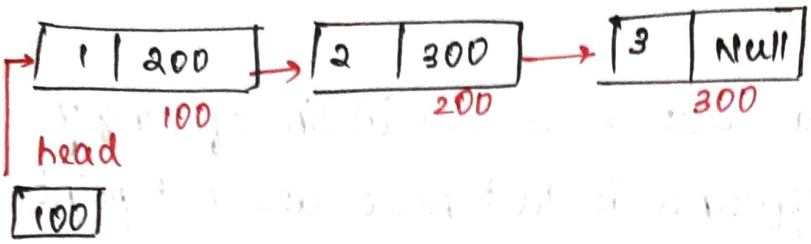
```
struct node
{
  int data;
  struct node next;
};
```

TYPES OF LINKED LIST:

1) SINGLY LINKED LIST:

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

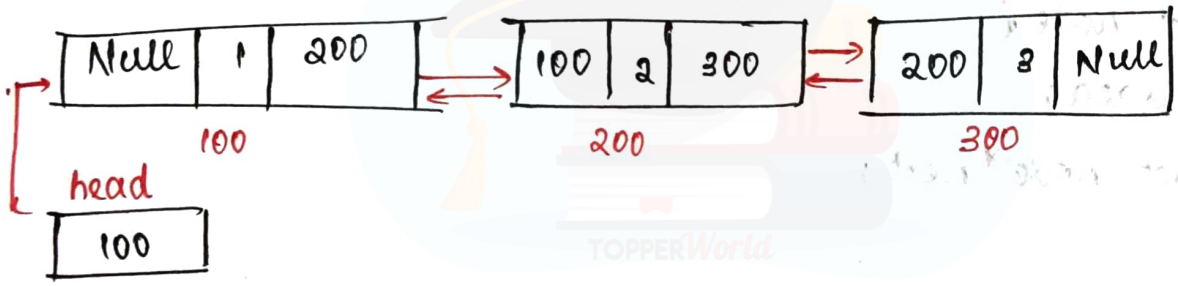
Suppose we have, three nodes and address of these three nodes are 100, 200 and 300:



Null means it address part does not point to any node. The pointer that holds address of initial node is known as header 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 two address part

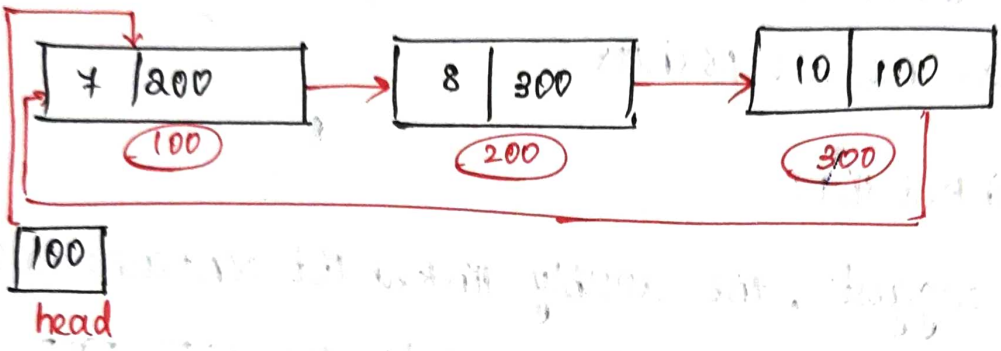


Representation of doubly linked list:

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

3) CIRCULAR LINKED LIST:

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

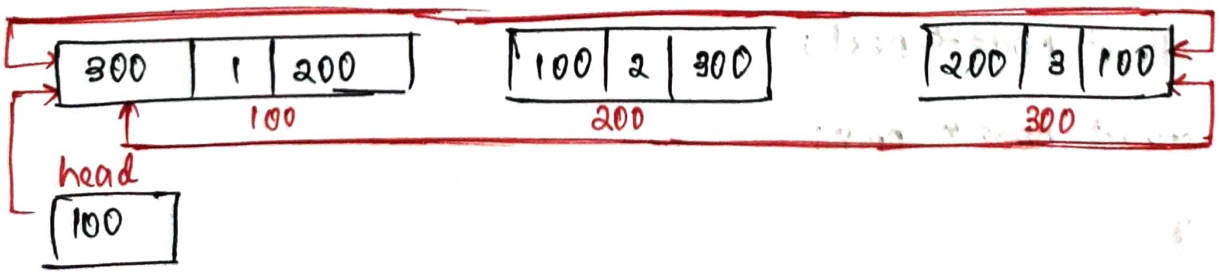


Representation of circular linked list:

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

4) Doubly circular linked list:

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



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

Representation of doubly circular linked list:

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


Complexity.

	Average				space complex
singly linked list	Access O(n)	search O(n)	insertion O(n)	deletion O(1)	worst O(n)
Singly linked list	worst access O(n)	search O(n)	insertion O(1)	deletion 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 of beginning: It involves inserting any elements at time while front of list. we just need a few link adjustment to make new node as head

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

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

② 3) Deletion and Traversing:

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

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

③ Traversing: visiting each node of the list atleast once in order to perform some specific operation on it, for example printer data part of each node present in the list

Searching: In searching, we match each element of the list with the given elements. If elements is found on any of location of that elements 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 beginning: Adding the node into the linked list at the beginning

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

3) Deletion and Traversing:

① Deletion at beginning: Removing the node from the beginning of the list

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

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

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

SKIP LIST:

What is a skip list?

skip list is used to store a linkedlist of elements or data with linked list. The one single step, it skip several elements of entire list.

STRUCTURE OF SKIPLIST:

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 expressline where elements are skipped.

COMPLEXITY TABLE:

SNO	COMPLEXITY	AVERAGE CASE	WORST CASE
1	Access complexity	o(log n)	o(n)
2	search complexity	o(log n)	o(n)
3	delete	o(log n)	o(n)
4	Insert	o(log n)	o(n)
5	space	—	o(n log n)

BASIC OPERATIONS AND ITS ALGORITHM:

① Insertion operation: It is used to add new node to 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]
a = L → header
for i = L → level down
    while a → Forward[i] → key forward [i]
update [i] = a
a = a → Forward [0]
lvl = random_level ()
if lvl > 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 ... max level+1]
a = L → header
for i = L → level down 0 to 90
```

```
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:

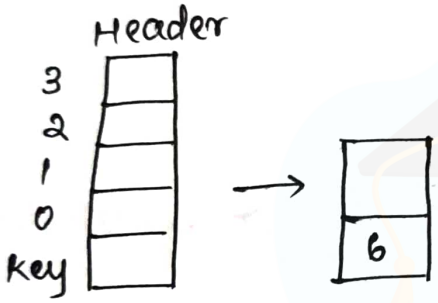

```
Searching (L, skey)
    a = L → header
    loop invariant : a → key level down to 0 do
    while a → Forward [i] → key forward [i]
    a = a → Forward [a]
    if a → key = skey then return a → value
else return failure.
```

Example:

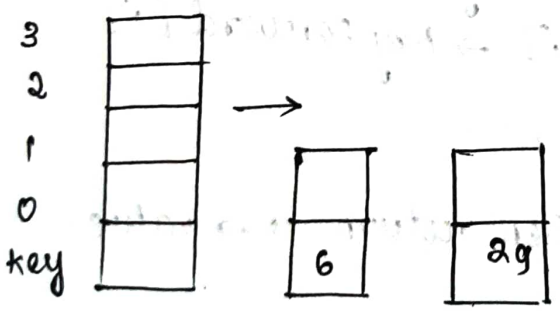
Create a skip list, we want to Insert these following keys in empty skip list.

1. 6 with level 1
2. 2g with level 1
3. 22 with level 4
4. g with level 3
5. 17 with level 1
6. 4 with level 2

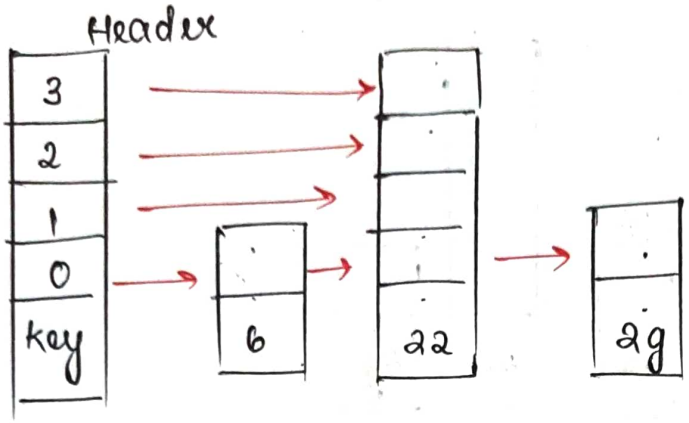
solution: Insert 6 with level 1



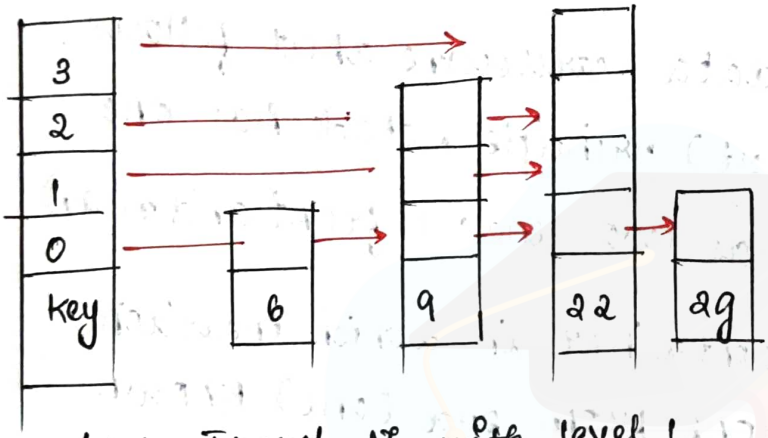
Step 2: Insert 2g with level 1



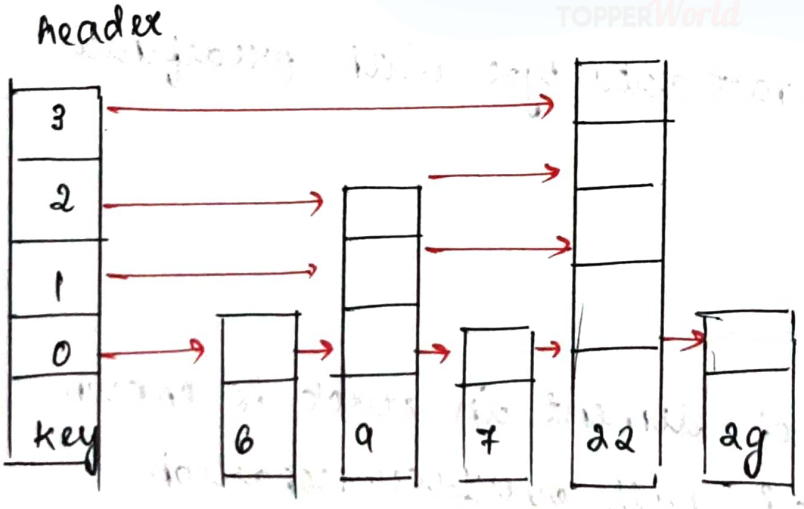
Step 3: Insert 22 with level 4



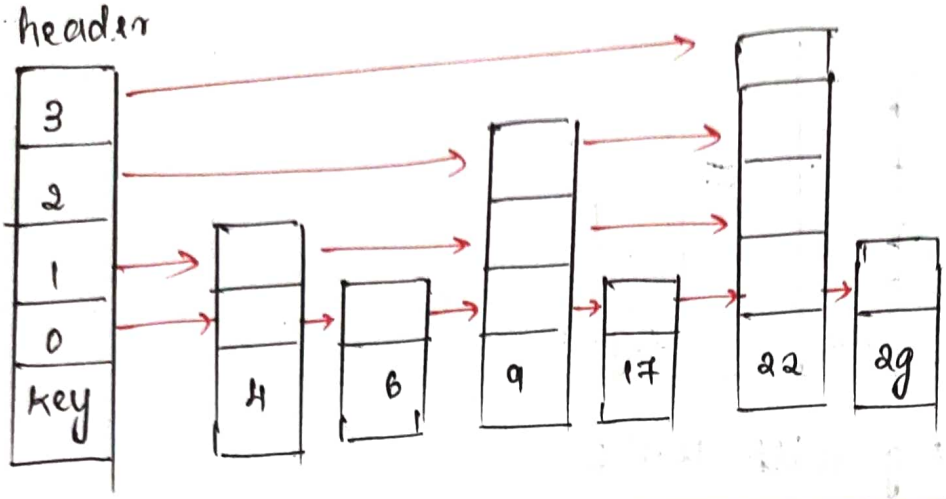
Step 4: Insert 9 with level 3



Step 5: Insert 17 with level 1



step6: Insert 4 with level 2



STACK:

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 end (one) known as top of stack.

A stack is abstract data type with predefined capacity.

OPERATIONS OF STACK:

push() - insertion of element in stack is known as push. If stack is full overflow condition occurs.

pop( ) Deletion of element from stack is known as pop operation. If stack is empty, underflow occurs

peek( ) It returns element at given index

count( ) It returns total number of count

change( ) It changes the element at a given position

display( ) It prints all the elements available.

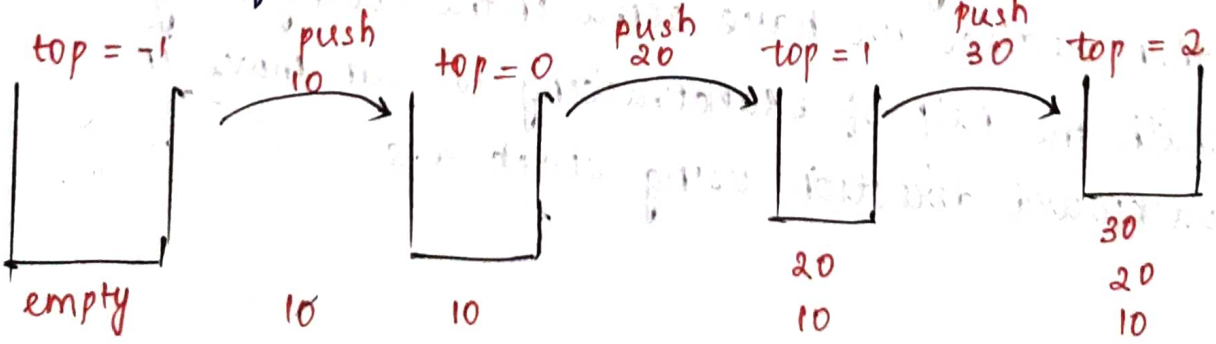
PUSH OPERATION:

Before inserting element in stack, we check whether stack is full.

If stack is full and we try to insert, overflow condition occurs.

we set top value as -1. to check whether stack is empty

The elements will be inserted until we reach max size of stack ie top = top+1



POP OPERATION:

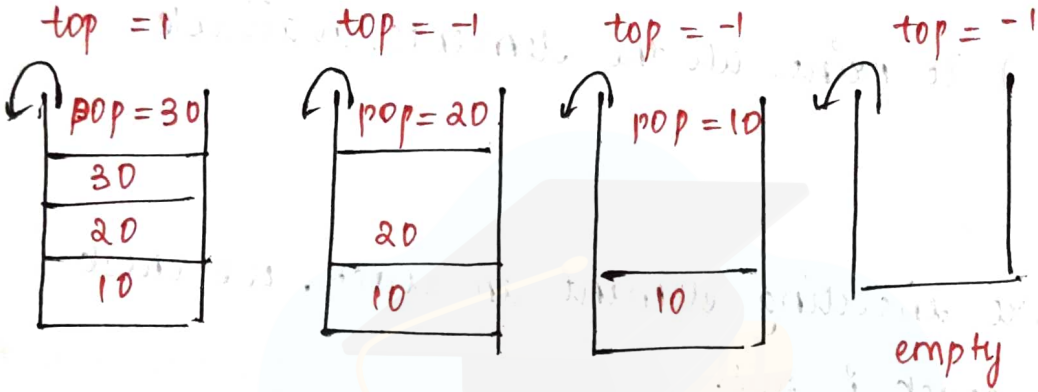
Before deleting we check whether stack is empty

If stack is empty, we try to delete elements, underflow condition occurs.

First access element which is pointed by top

Once top is performed, top is decremented by 1

ie top = top - 1



APPLICATIONS OF STACK:

Recursion: The recursion is a method calling itself again and again

Depth First search: This is implemented on a graph using stack d.s

BackTracking: If we move in forward direction & then realise we have come on the wrong way with the help of backtracking we can move in backward direction using stack d.s

4) Memory management: The stack manages the memory The memory is assigned in continuous memory blocks.

Algorithm

Push Operation

```
begin
  if top = n, stack is full
  top = top+1
  stack(top) = item
end
```

Time complexity: O(1)

pop operation

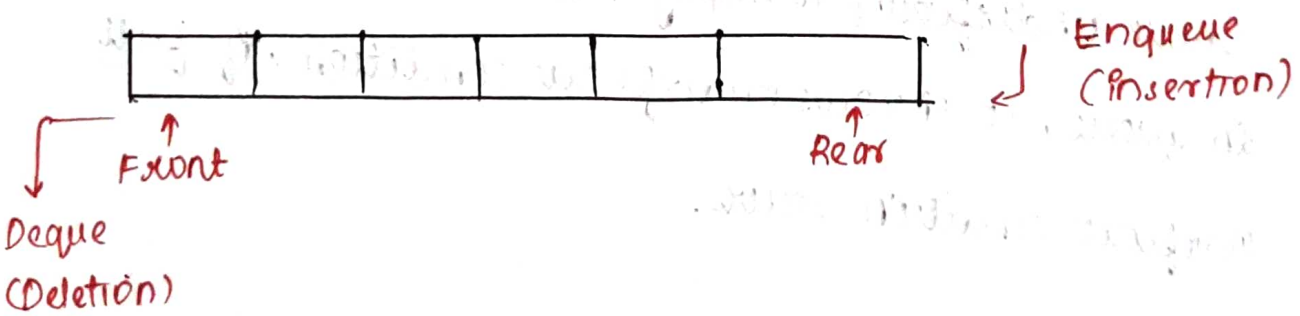
```
begin
  if top = 0, empty
  item = stack(top);
  top = top-1;
end
```

Time complexity: O(1)

QUEUE:

A queue can be defined as ordered list which enables insert operations to be performed at oneend called REAR and delete operations to be performed at another end called FRONT

Queue can be referred as: to be first in first out.



COMPLEXITY OF QUEUE:

Queue	Average Access O(n)	search O(n)	Deletion O(1)	insertion O(1)	space complex worst O(n)
Queue	worst Access O(n)	search O(n)	Deletion O(1)	insertion O(1)	

OPERATIONS ON QUEUE:

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

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

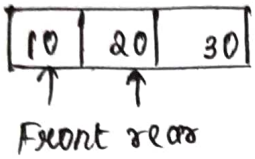
Peek: This returns, element which is pointed by front pointer in the queue but does not delete it.

Queue overflow (Is Full): when queue is completely full then it shows overflow condition

Queue underflow (Is empty): when there is no element in queue, it throws underflow condition, If full overflow condition occurs.

TYPES OF QUEUE:

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.

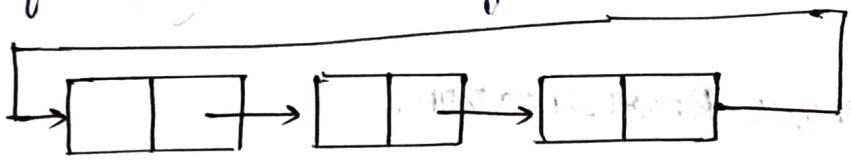


The elements are inserted from rear end, and if we insert more elements in queue, then rear values gets incremented on every insertion.

Drawback of linear queue is: insertion is done only from rear end.

Circular Queue

In circular queue, the last element points to the first element making a circular link

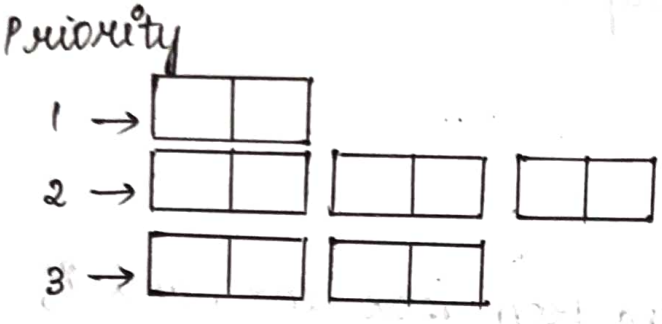


Circular Queue Representation.

The main advantage of circular queue over a simple queue is better memory utilization. If the last position is full and first position is empty. we can insert in first position. This is not possible in simple queue

Priority Queue:

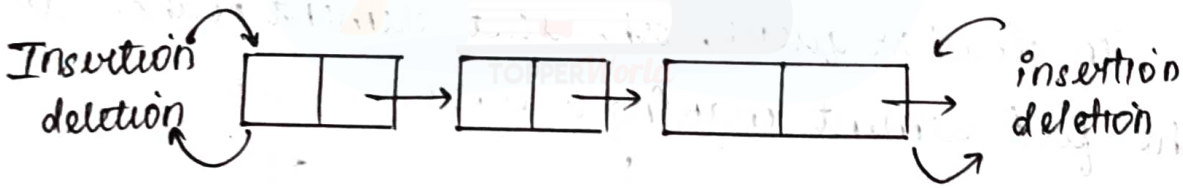
It is a type of queue served according to its priority. If elements with same priority occur, they are served according to their order in queue.



Priority Queue Representation

Deque (Double Ended Queue)

In a double ended queue, insertion and removal of elements can be performed from either from front or rear. It does not follow FIFO.



Deque Representation.

ALGORITHM TO INSERT ANY ELEMENT IN A QUEUE:

check if queue is already full by comparing rear to max - 1.

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

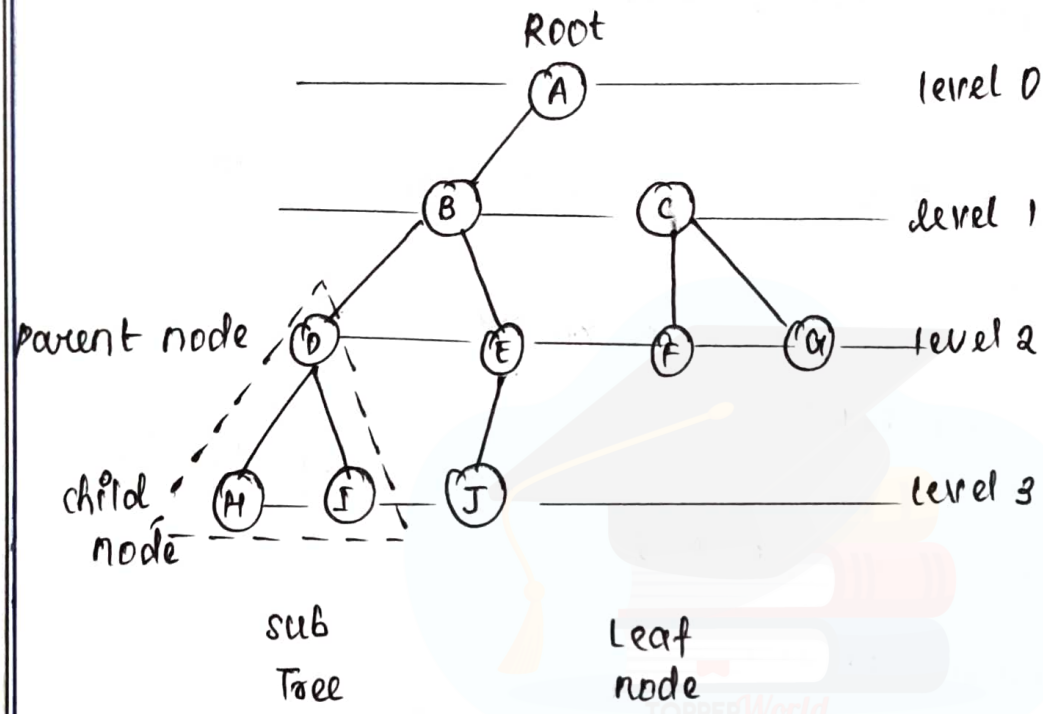
ALGORITHM TO DELETE AN ELEMENT FROM QUEUE

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:

A tree is a non linear abstract datatype with hierarchy based structure. It consist of nodes that are connected via links. The tree data structure stems from a single node called root node and has subtrees connected to root



IMPORTANT TERMS:

Following are the important terms with respect with Tree

path: Path refers to sequence of nodes along the edge of a tree

Root: The node at top of tree is called root.

Parent: Any node except root node has one edge upward to node called parent

Child: The node below the given node connected by its edge downward is called its child node.

Leaf: The node which does not have any child node is called leaf node

Subtree: Subtree represents descendants of node

Visiting: visiting refers to checking the value of node when control is on node.

Traversing: Traversing means passing through nodes in specific order

Levels: Levels of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1. its grandchild is at level 2 and so on.

Keys: Key represents a value of a node based on which a search operation is to be carried out for a node.

Types of Trees:

There are three types of trees:

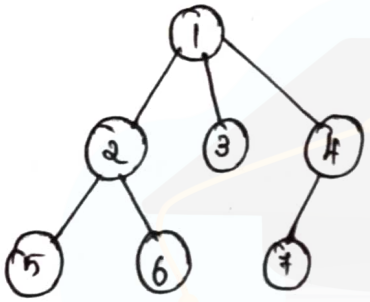
General Trees

Binary Trees

Binary Search Trees.

General Trees:

General Trees are unordered tree data structure where the root node has minimum 0 or maximum 'n' subtrees.



General Tree Data Structure

Binary Trees:

Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees. left subtree and right subtree.

Full Binary Tree:

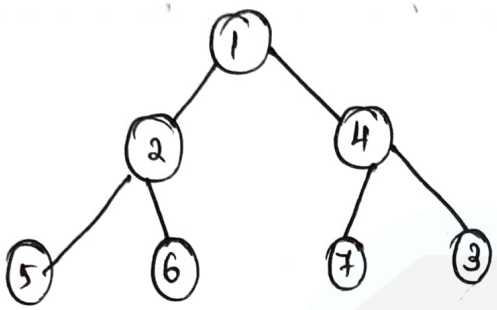
A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.

Complete Binary Tree:

A complete binary Tree is a binary tree where all leaf nodes must be on same level.

Perfect Binary Tree:

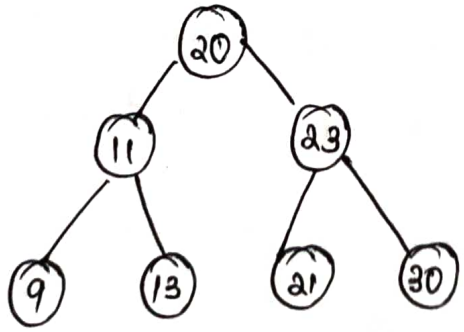
The leaf nodes are on same level and every node except leaf nodes have 2 children



Binary Tree data structure

Binary Search Tree

The data in Binary search Tree is always stored in such a way that the values in the left subtree are always less than values in root node and the values in the right subtree are always greater than values in root node.



Binary Search Tree Data Structure

Balanced Binary Search Tree:

Consider a balanced binary search tree with 'm' as height of left subtree and 'n' as height of right subtree. If the value of (m-n) is equal to 0, 1, or -1, the tree is said to be Balanced Binary Search Tree.

There are various types of self-balancing binary search Trees

- AVL Trees
- Red Black Trees
- B Trees
- B+ Trees
- Splay Trees
- Priority Search Trees

GRAPH:

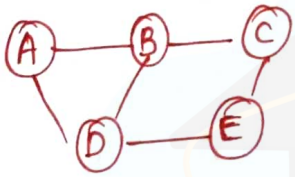
A graph can be defined as group of vertices and edges that are used to connect these vertices.

Let us try to understand this through example: on facebook, everything is a node. That includes user, photo, album, event, group, page, comment, story. anything that has data is a node.

Example of graph data structure.

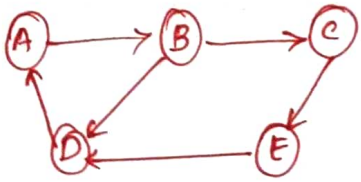
Directed and Undirected graph:

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



Undirected graph

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



Directed graph

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

GRAPH TERMINOLOGY

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

Closed Path: A path will be called as closed if initial node is same as terminal node $v_0 = v_N$

Simple path: If all nodes of graph are distinct with an expression $v_0 = v_N$, then such path is called as closed simple path

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

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.

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

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

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

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

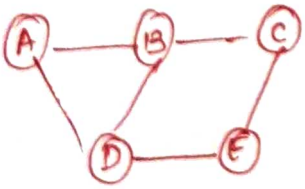
Adjacent nodes: If two nodes u and v are connected via an edge e, then nodes u and v are called as neighbours.

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

GIRAPH REPRESENTATION:

SEQUENTIAL Representation:

In this we use adjacent matrix to store mapping represented by vertices and edges. A graph having n vertices will have dimension n×n.



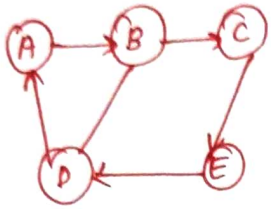
undirected graph.

	A	B	C	D	E
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

Adjacency matrix.

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

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

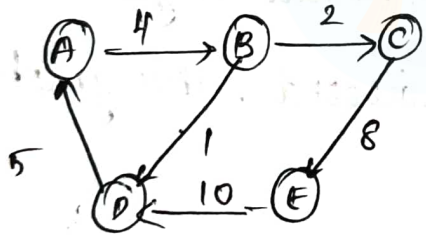


Directed graph.

	A	B	C	D	E
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

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 edges

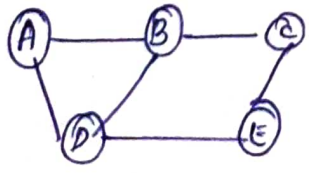


Weighted undirected graph

	A	B	C	D	E
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

Adjacency matrix.

② Linked representation:



undirected graph

Adjacency list.

A → B → D X

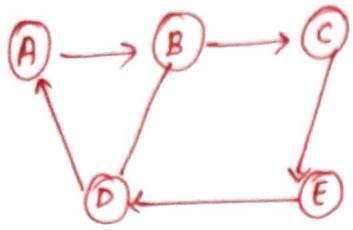
B → A → D → C X

C → B → E X

D → A → B → E X

E → D → C

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



Directed graph

A → B ×
B → C → D ×
C → E ×
D → A ×
E → D ×

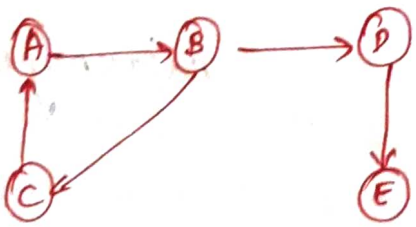
In directed graph, sum of lengths of all adjacency lists is equal to number of edges present in the graph.

GRAPH TRAVERSAL ALGORITHM:

The graph is one non-linear data structure. That is consist of some nodes and their connected edges.

The edges may be directed or undirected. This graph can be represented as G(V, E)

The following graph can be represented as G({A,B,C,D,E}, {(A,B), (B,D), (D,E), (B,C), (C,A)})



The graph has two traversal algorithms.

Breadth First Search

Depth First Search.

Breadth First Search (BFS)

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

Then it selects nearest nodes and explores all unexplored nodes.

The algorithm follows same process for each of nearest node until it finds goals.

Algorithm:

step 1: SET STATUS = 1 (ready state)
for each node in

step 2: enqueue starting node A & set its status = 2 (waiting) states

step 3: repeat step 4 and 5 until queue is empty

step 4: Dequeue a node N. process it. set its STATUS = 3

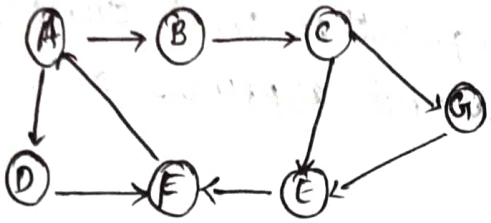
step 5: Enqueue all neighbours of N that are in ready state (whose STATUS = 1) & set (STATUS = 2)

[END OF LOOP]

step 6: Exit

EXAMPLE

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



Adjacency list

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

Solution: minimum path p can be found by applying Breadth First Search algorithm that will begin at node A and will end at node E

$$A \rightarrow B \rightarrow C \rightarrow F$$

Depth First Search Algorithm:

DFS starts with initial node of graph g and goes to deeper and 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

step 4: pop top node N. process it & set its STATUS = 3

step 5: push on stack all neighbours of N that are in readystate (whose status = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]

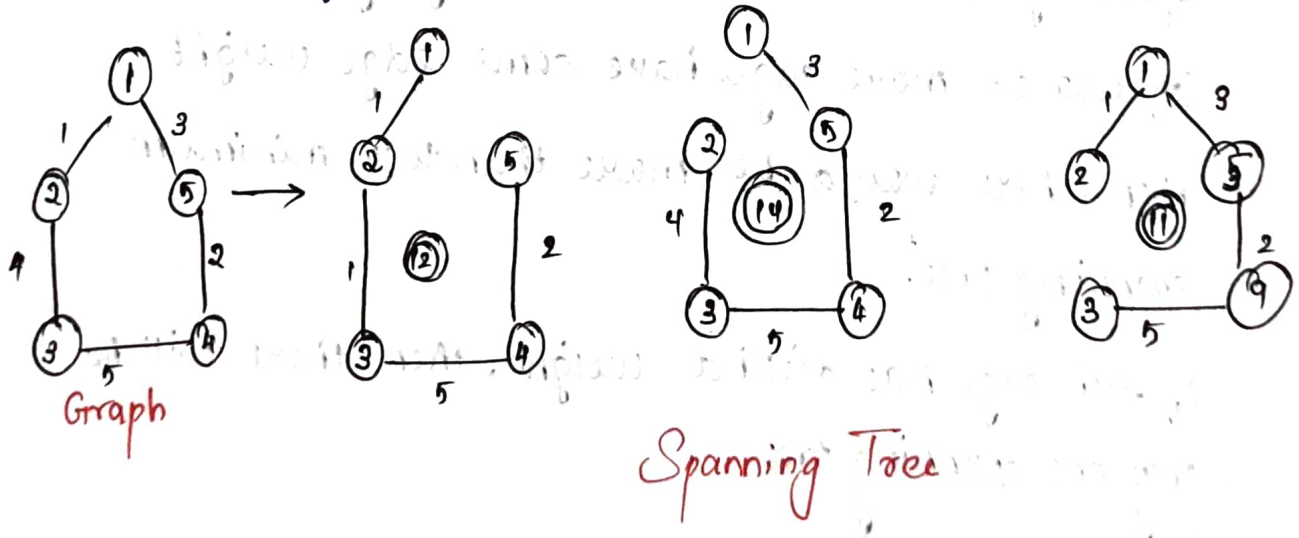
step 6: EXIT

SPANNING TREE:

If we have a graph containing v vertices and E edges, then the graph can be represented as G(V, E)

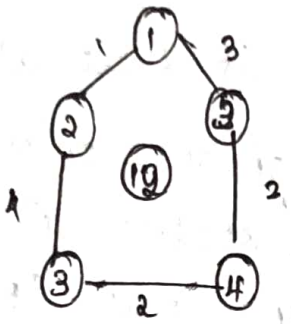
If we create spanning tree from above graph then spanning tree would have some number of vertices as the graph. but vertices are not edges

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



Minimum Spanning Trees:

The minimum spanning Tree is a tree whose sum of edges weights is minimum.



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

Properties of spanning Tree:

A connected graph can contain more than 1 spanning tree.

All possible spanning trees that can be created from given graph minus 1

spanning trees does not contain any cycle

If two or more edges have some edge weight then there would be more than two minimum spanning tree.

If each edge has distinct weight, then there will be only one spanning tree

Applications of spanning tree:

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

Clustering - grouping set of objects in such a way that similar objects belongs to same group than to different group.

SEARCHING:

There are two methods widely used:

Linear Search
Binary Search.

ALGORITHM FOR LINEAR SEARCH.

LINEAR_SEARCH (A, N, VAL)

step 1: [INITALIZE] SET POS = -1

step 2: [INITALIZE] SET I = 1

step 3: Repeat step 4 and 5

step 4: If A[I] = VAL

SET POS = I

PRINT POS

GO TO STEP 6

[END OF IF]

PRINT POS

Goto 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 ALGORITHM:

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

C PROGRAM

LINEAR SEARCH

```
#include<stdio.h>
void main()
{
int a[10]={10,23,40,1,2,0,14,13,50,99};
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");
 }
}
```


Output:

Enter item which is to searched:

20

Item not found

Enter item which is to be searched:

23

Item found at location 2.

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 with BEG <= ENG

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

SNO	PERFORMANCE	COMPLEXITY
1	WORST CASE	O(log n)
2	BEST CASE	O(1)

3) Average case — O(log n)

4) worst case — O(1)
space complexity

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

In 1st step

BEG = 0

END = 8

MID = 4

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

In 2nd step

Beg = 8 mid + 1 = 5

End = 8

mid = 13/2 = 6

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

In 3rd step

Beg = mid + 1 = 7

End = 8

mid = 15/2 = 7

a[mid] = a[7]

a[7] = 23 = item

∴ SET LOCATION = mid.

location of item is 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

step 3

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 binarySearch (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 item 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 binarysearch (a,mid+1, end, item);
  }
  else{
  return binary search (a, beg, mid -1, item);
  }
 }
return -1
}
```

Output:

enter item to search

19

item found at location 2

SORTING ALGORITHM

BUBBLE SORT

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)

case	Time complexity	space complexity
Average case	$O(n^2)$	
worst case	$O(n^2)$	

Implementation of Bubble sort

```
#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[j]= a[j];
    a[j] = temp;
    }
   }
  }
}
 void main(){
     int i, j, temp;
     int a[5] ={10, 25, 32, 13, 25};
     int n= sizeof(a)/ size of (a0);
     printf("before sorting array elements");
     prtbf (a,b);
     bubble (a,n);
     printf("after sorting ");
     printf (a,n);
     }
```

Output

Before sorting array elements

10 35 32 13 26

after sorting

10 13 26 32 35

BUCKET SORT ALGORITHM

Bucket sort (A[])

1. let B[0...n-1] be 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] while insertion sort
9. concatenate list B[0], B[1] ... B[n-1] together in order
10. END.

COMPLEXITY:

Time Complexity

case	Time complexity
Best case	o(n+k)
Average case	o(n+k)
worst case	o(n²)

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(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];
for (i=0; i<=max; i++)
  {
    bucket[i] = 0;
  }
for(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");
printfArr (a, n);
bucket (a, n);
printf ("after sorting");
printArr (a, n);
}
```

Output

Before sorting

54 12 84 57 69 41 9 5

after sorting

5 9 12 41 54 57 69 84

HEAP SORT ALGORITHM:

```
Heap Sort (arr)
Build MaxHeap (arr)
for i = length (arr) to 2
```

Swap arr[i] with arr[i]
heap_size[arr] = heap_size[arr]?!
MaxHeapify(arr, i)

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>
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--)
  {
  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");
  printArr(a,n);
   heapsort (a,n);
   printf (" \n after sorting ");
   prin tf (a,n);
   return 0;
 }
```


O/P

Before sorting

48 10 23 43 28 26 1

after sorting

1 10 23 26 28 43 48

INSERTION SORT

COMPLEXITY

case-time complexity	
Best case $O(n)$	space complexity $O(1)$
Average case $\Theta(n^2)$	
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 = sizeof (a)/ sizeof (a[0]);
  printf (" before sorting");
  printArr (a,n);
  insert (a,n);
  printf ("after sorting");
  printArr (a,n);
  return 0;
}
```


output

Before sorting

12 31 25 8 32 17

after sorting

8 12 17 25 31 32

MERGE SORT ALGORITHM:

```
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

```
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];
  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++;
 }
}
```

DATA STRUCTURE CODING QUESTION

① ARRAYS USING C:

Program to demonstrate arrays in C.

```
#include <stdio.h>
#include <window.h>
#include <stdlib.h>
#define NUM_EMPLOYEE 10.
int main (int arg c, 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", gcount);
```

```
printf("There are %d employee with salary less than
        3000\n", dcount);
printf("press enter to continue...\n");
getchar();
return 0;
}
```

② 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 <TY * tail_ = NULL;
Node <T> * ptr = NULL;
public;
Linked list()
{
this -> size_ = 0;
 void append (T value)
 {
 if (this -> head == NULL)
 {
 this->head_= new Node <TY (value);
 this -> tail = this -> head -;
 }
 else{
 this -> tail_ -> next = new Node <TY (value)
 this -> tail_ -> next = previous = this_ -> tail;
 }
 this -> size_t = 1;
 }
 void append (T value)
 {
 void resetIterator()
 {
 tail_ = Null;
 }
int main (Int arg c, char ** arg v)
 {
```

```
linked list <int> llist;
 llist . append (10);
 llist . append (3);
cout <<" printing linked list << endl;
cout << endl;
return 0;
```

Stack Implementation in C:

```
#include <stdio.h>
int MAXSIZE =8;
int stack[8];
int top =-1;
int isempty (){
   if (top== =-1) {
      return1;
    else
     return 0;
    }

int isfull (){
   if (top == MAXSIZE)
    return1;
   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 retrieve 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 onto the stack
        push(3);
        push(5);
```

```
push(9);
push (1);
push (12);
push(15);
printf("elements at top of stack: %d\n", peek());
printf("Elements :\n");
//print stack data.
 while (! isempty()){
 int data = pop();
 printf("%d\n", data);
}
printf("stack is full: %s\n", is full() ? "true" : "false");
printf("stack is empty: %s\n", is empty() "true", "false");
return 0;
}
```

DATA STRUCTURE INTERVIEW QUESTIONS:-

① What is a Data structure:

A data structure is a logical way that data is organised within a program

② Why create Data Structure?

Data structure serve number of functions (important) in a program

③ what are some applications of Data structure

Block chain

Database Design

Compiler Design

Genetics

Decision making

Numerical and stastical analysis.

④ Describe types of data structure?

Linear Data structure

Non-linear Data structure

⑤ what are the different operations available in stack?

push
pop
top
isEmpty
size

⑥ what are the application of Queue?

call management in call centres
Operating system: job scheduling operations, CPU scheduling

⑦ what are the different operations available in dequeue?

enqueue
dequeue
isEmpty
rear
front
size

⑧ Difference b/w stack and Queue.

stack is based on LIFO
Queue is based on FIFO.
stack is used in solving recursion
Queue is used in sequential processing problems.

⑨ what are the methods available in storing sequential files?

Straight merge
natural merge
polyphase sort
distribution of initial runs.

(10) list out few of applications of tree data structure?

The manipulation of arithmetic expression, symbol table construction & syntax analysis

(11) How to check whether linked list is circular?

create two pointers, each set to start of list

update each as follows:

```
while(pointer)
  {
  pointer 1 = pointer 1 → next;
  pointer 2 = pointer 2 → next;
  if (pointer 2) pointer 2 = pointer 2 → next;
  if (pointer 1 == pointer 2)
  {
  print ("circular")
  }
  }
```

(12) List the basic Operations carried out in linkedlist?

creation of list
Insertion of list
deletion of list
modification of node
Traversal of node

(13) list out the basic operations that can be performed on stack.

push
pop
peek
empty
fully occupied.

(14) State the different ways of representing expression

Infix notation
prefix notation
postfix notation

(15) what is sequential search?

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

(16) what are the different types of linkedlist?

Singly linked list
doubly linked list
circular linked list