Data Structure

- The group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.
- A way of arranging data on a computer so that it can be accessed and updated efficiently.
- Examples: arrays, Linked List, Stack, Queue, Tree, Graph etc.

Types of Data Structure

There are two types of data structures:
1. Primitive data structure
2. Non-primitive (Abstract) data structure

1. Primitive Data structure (type)
The primitive data structures are primitive data types.
Example: int, char, float, double, and pointer are the primitive data structures that can hold a single value.

2. Non-Primitive Data structure:-
- The non-primitive data structure is also called abstract data Structure.
- It is divided in two types:

a) Linear data structure:- The arrangement of data in a sequential manner is known as a linear data structure.
For example:- Arrays, Linked list, Stacks, Queues and hashing.

b) Non-linear data structure:- This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.
For example:- Tree, Graphs.

Operation performed on data structure

1. Accessing
2. Searching
3. Sorting
4. Insertion
5. Deletion
6. Updation

Advantages of Data structures

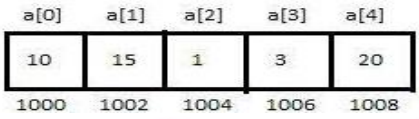
- Efficiency
- Reusability
- Abstraction

1. Array

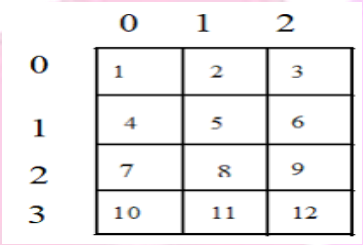
- An array is a linear and homogeneous data structure.
- An array is a collection of elements and each element is identified by an array index or key.
- It means that similar types of elements are stored contiguously in the memory under one variable name.
- An array can be visualized as a row in a table, whose each successive block can be thought of as memory bytes containing one element.
- The size of an array is specified at the time of its creation and cannot be changed dynamically.
- Arrays are zero-indexed in C,C++, java etc, i.e. the first element is accessed with index 0 and last element is (size-1).
- Array can be accessed randomly using their index, which allows for constant-time complexity O(1) for read and write operations.
- We can initialize array at compile time and run time also.

Types of Array

1. **Single Dimensional Array / One Dimensional Array**



2. **Multi Dimensional Array**



	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

1. Single Dimensional Array / One Dimensional Array

Declaration and Initialization of 1D Array

- **C/C++:** int arr[5]; for declaration and int arr[5] = {1, 2, 3, 4, 5}; for initialization.
- **Python:** arr = [1, 2, 3, 4, 5] (lists in Python can act as arrays).
- **Java:** int[] arr = new int[5]; for declaration and int[] arr = {1, 2, 3, 4, 5}; for initialization.

1 D Array address

To find the address of a elements in array or to access an array element direct:

Address of an element (arr[i]) = Base address + index * size of data_type

Two Dimensional Array

- Array having more than one subscript variable is called Multi-Dimensional array.
- Multi Dimensional Array is also called as Matrix.

Syntax:

<data type> <array name> [row subscript][column subscript];

Example:- int arr[m][n]; where m is the row number and 'n' is the column number.

2D array is also called array of arrays. Because every row is considered as array and there are multiple rows in 2 D arrays.

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Declaration of 2-D array

- **C/C++:** int arr[3][2];
  char arr[][3]; // we can skip the size of row subscript but we cannot skip the size of column subscript.
- **Python:** Lists of lists can be used to create 2D arrays,
  arr = [[1, 2], [3, 4]].
- **Java:** int[][] arr = new int[3][4];.

Initializing Two–Dimensional Arrays

Compile time initialization

int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};

Run time initialization:

```
int arr[3][3],i,j;
   for (i=0;i<3;i++)
   {
      for (j=0;j<3;j++)
      {
         printf("Enter arr[%d][%d]:\n ",i,j);
         scanf("%d",&arr[i][j]);
      }
   }
```

Accessing Elements of Two-Dimensional Arrays

Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.
Example: consider an 2 D array

In this array matrix value of the index arr[2][1] = 29

65 (0,0)	37 (0,1)	8 (0,2)	12 (0,3)
14 (1,0)	25 (1,1)	36 (1,2)	27 (1,3)
48 (2,0)	29 (2,1)	10 (2,2)	61 (2,3)

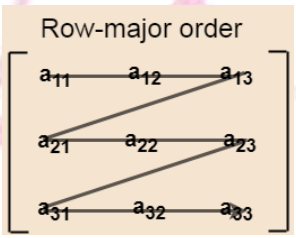
Example:- int a[3][3]={4,9,5,6,7,1,2,3,8};
Printf("%d", a[1][2]);

Mapping 2D array to 1D array

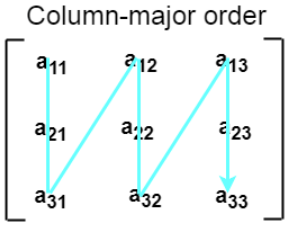
To store 2D array in the memory.

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

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



Column Major ordering:-
According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously.



Calculating the Address of the random element of a 2D array

If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as:

By Row Major Order:-

address(a[i][j]) = Base Address + (i*n + j) * SIZE

By Column major order:-

Address(a[i][j]) = base address +((j*m)+i)*Size

Advantages of an Array

- Accessing an element is very easy by using the index number.
- The search process can be applied to an array easily.

Disadvantages of an Array

- Fixed size can lead to wasted memory or insufficient space.
- The array stores only homogeneous data.
- The array stores data in contiguous (one by one) memory
- Location and memory suffers with external fragmentation.
- Insertion and deletion operations can be costly, to insert or delete shifting of the elements is required.

Time Complexity of Array on multiple operation

SNo.	Operation	Best	Average	Worst
1.	Accessing	O(1)	O(1)	O(1)
2.	Searching(linear search)	O(1)	O(n)	O(n)
3.	Searching(binary Search)	O(1)	O(logn)	O(logn)
4.	Insertion	O(1)	O(n)	O(n)
5.	Deletion	O(1)	O(n)	O(n)
6.	Updation	O(1)	O(1)	O(1)

Linked List

- To overcome insertion deletion and static memory allocation problem of array we use Linked List.
- Linked list is a linear data structure that includes a series of connected nodes.
- Linked list can be defined as the connected nodes that are randomly stored in the memory.
- A node in the linked list contains two parts, i.e., first is the data part and second is the address part.
- The last node of the list contains a pointer to the null.
- Memory is allocated at run time.

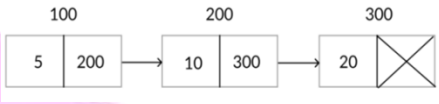
Types of linked list

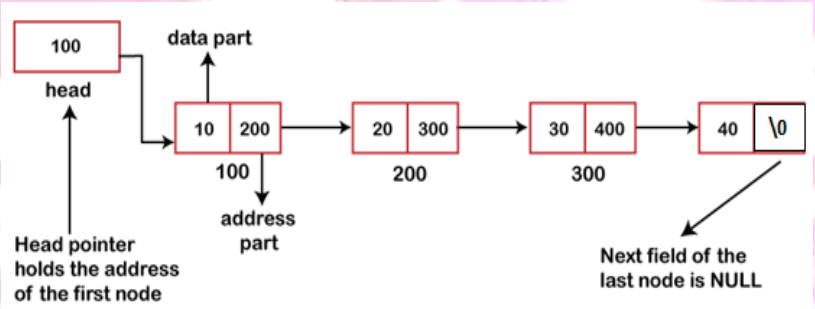
There are 4 types of linked list:-

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

1. Singly linked list

- Singly linked list can be defined as the collection of ordered set of elements.
- The number of elements may vary according to need of the program.
- A node in the singly linked list consists of two parts: data part and link part.
- Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.
- In Singly Linked List, only the pointer to the first node is stored. The other nodes are accessed one by one.
- To get the address of ith node, we need to traverse all nodes before it because the address of ith node is stored with i-1th node and so on.
- In other words, we can say that each node contains only next pointer, therefore we cannot traverse the list in the reverse direction.
- One way chain or singly linked list can be traversed only in one direction.





Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below:

1. **Node creation:-** we can create a node by using dynamic memory allocation
2. **Insertion:-** we can insert a node at the beginning , at the end, at the given position and after and before the given position
3. **Deletion:-** we can delete a node at the beginning, at the end and at the given position.
4. **Display:-** We can traverse each node and display the data of each node.
5. **Reversing a list:-** we can reverse a list by reversing the next pointers (address) of the node.
6. **Searching:-** we can search a list for a data.

Example:-

```
// Node structure
 struct Node {
   int data;
   Node* next;
      Node(int val) {
      data = val;
      next = nullptr;
   }    };
void insertAtPos(int data, int pos) {
  Node* newNode = new Node(data);
            if (pos == 0) {
         newNode->next = head;
         head = newNode;
         return;       }
```

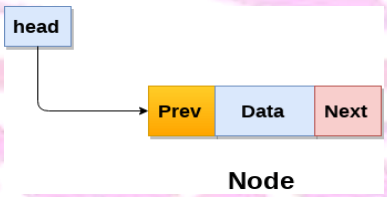
```
   Node* temp = head;
for(int i = 0;i < pos-1&& temp!=nullptr; i++)
 {          temp = temp->next;   }

if (temp == nullptr) {
cout <<"Position outof bounds"<<endl;
         delete newNode;
         return;
      }
       newNode->next = temp->next;
      temp->next = newNode;
   }
```

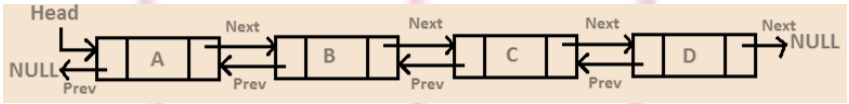
Complexity of singly linked list

Data Structure	Time Complexity	Space Complexity
Singly Linked List	**Worst**	**Worst**
Access	O(n)	O(n)
Search	O(n)	
Insertion	Beginning: O(1), End: O(n) At position: O(n)	
Deletion	Beginning: O(1), End: O(n) At position: O(n)	
Reverse	O(n)	
Search	O(n)	

2. Doubly linked list

- The major drawback of the singly Linked List is that it is not easy to go to the previous element to overcome this problem we use Doubly linked list.
- In a doubly linked list, a node consists of three parts:
  1. Pointer to the previous node (previous pointer),
  2. data and
  3. Pointer to the next node in sequence (next pointer).
- In doubly Linked List transversal is possible in both the directions, forward and backward easily as compared to the Singly Linked List.
- The First Element will have its previous pointer as Null, to mark the start of the List.
- The Last Element will have its Next pointer as Null, to mark the end of the List.





Operations on Doubly Linked List

1. Node creation
2. Insertion
3. Deletion
4. Display
5. Traversal (we can perform forward and backward traversal.)
6. Reverse
7. Search

Advantages of DLL over singly linked list

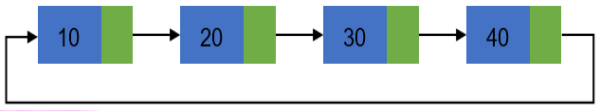
1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.
4) In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages of DLL over singly linked list

a. Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See this and this).
b. All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Circular linked list

- When we want to access the things in a loop repeatedly. Circular Linked List is ideal so that, whenever we reach the last node we can restart operations again by directly reaching the first node from the last node itself.
- Creating a circular linked list is no different from creating a singly linked list. One thing we do differently is that instead of having the last element to point to NULL, we'll make it point to the head.



- Circular Linked List is a variation of Linked list in which the first element points to the last element to form a circle.
- We traverse a circular singly linked list until we reach the same node where we started.
- There is no null value present in the next part of any of the nodes.
- A circular linked list can be a singly circular linked list or a doubly circular linked list.

Operation on circular linked list

We can perform the following operations on a circular linked list.

1. Node creation
2. Insertion
3. Deletion
4. Reversing
5. Traversing
6. Searching

Circular doubly linked list

- Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node.
- Circular doubly linked list doesn't contain NULL in any of the node.
- The last node of the list contains the address of the first node of the list.

- The first node of the list also contains address of the last node in its previous pointer.

Example

```
#include <iostream>
using namespace std;
// Node structure
struct Node {
   int data;
   Node* next;
      Node(int val) {
      data = val;
      next = nullptr;
   }      };
// LinkedList class
class LinkedList {
private:
   Node* head;
public:
   // Constructor
   LinkedList() {
      head = nullptr;    }
// Destructor
   ~LinkedList() {
      Node* current = head;
      Node* next;
            while (current != nullptr) {
         next = current->next;
         delete current;
         current = next;
      }
  head = nullptr;    }
// Insert at the beginning
   void insertAtBeginning(int data) {
      Node* newNode = new Node(data);
      newNode->next = head;
      head = newNode;
}
// Print the linked list
void printList() const {
   Node* temp = head;
   while (temp != nullptr) {
      cout << temp->data << " -> ";
      temp = temp->next;    }
   cout << "NULL" << endl;  }   };
     int main() {
        LinkedList list;
        list.insertAtBeginning(8);
       cout << "Linked list: ";
        list.printList();
          return 0;           }
```

Drawbacks of Linked list

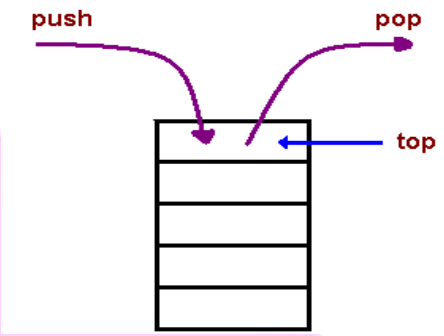
- Random access is not allowed. We have to access elements sequentially starting from the first node(head node). So we cannot do binary search with linked lists efficiently with its default implementation.
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Difference between array and linked list

S.No.	Array	Linked List
1.	An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address.
2.	Array elements store in a contiguous memory location.	Linked list elements can be stored anywhere in the memory or randomly stored.
3.	Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
4.	Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
5.	In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
6.	We can direct access of data.	We can sequentially access the data not direct.
7.	Accessing any element in an array is faster as the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
8.	Cost of inserting an element at the start is high O(n)	Cost of inserting an element at the start is low O(1)
9.	Cost of inserting an element at the end is low O(1)	Cost of inserting an element at the end is high O(n)
10.	Cost of inserting an element at the ith position or mid position is approximately same as linked list O(n)	Cost of inserting an element at the ith position or mid position is approximately same as array O(n)
11.	The implementation of an array is easy as compared to the linked list.	Comparatively complex
12.	The array is static or fixed in size	The linked list is dynamic in size
13.	Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement.
14.	Memory requirement of array is less than linked list as it store only data.	Memory requirement of linked list is more than array as it store only data and address both in a node.

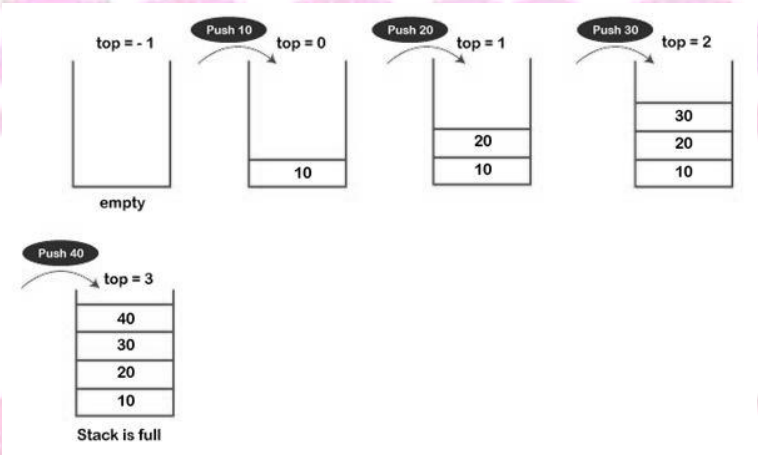
Stack

- A Stack is a non primitive linear data structure that follows the LIFO (Last-In-First-Out) or FILO (First-In-Last-Out) principle. Stack has one end which is called top.
- It contains only one pointer (top) pointing to the topmost element of the stack.
- Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
- The last item to be inserted into a stack is the first one to be deleted from it.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.
- In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.
- When the top element of a stack is deleted, if the stack remains non-empty, then the element just below the previous top element becomes the new top element of the stack.
- Example:- piles of books, a deck of cards, piles of plates etc.



Basic Operations on Stack

Push Operation:- When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs. We have only one end to insert a unique element on top of the stack, it inserts the new element at the top of the stack.



We can implement Push operation in 2 ways

a) Array implementation of stack

b) Linked list implementation of stack

Array implementation of push operation

- We can push element only in one side of array using top variable (pointing to current position) and increment the top every time when we push next item.
- Array implementation of stack suffers from overflow condition.

Function to Push element into the stack using array

```
void push (int x)
{
    int maxsize=10, stack[maxsize];
    int top=-1;
    if ( top == maxsize-1 )
   {
 Cout<<"Stack is full. Overflow condition!" <<endl ;
    }
 else
   {
    top = top+1;
    stack[top]=x;
   }
 }
```

Linkedlist implementation of Push operation

- A stack is also implemented dynamically where the size of stack is not fixed. The memory is created whenever it is needed.
- Push operation is performed from the beginning of the linked list. This is equivalent to insertion at beginning in linked list operation. Here we use top pointer instead of head pointer.

Pop Operation

- When we delete an element from the stack, the operation is known as a pop.
- We have only one end pop an element called top of the stack.
- If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

We can implement Pop operation in 2 ways

a) Array implementation of stack
b) Linkedlist implementation of stack

Array implementation of Pop operation

- We can implement stack by using array.
- We can pop element only in one side of array using top variable (pointing to current position) and decrement the top every time after we pop the item.
- Array implementation suffers from underflow condition when stack is empty.

Function to Pop element into the stack using array

```
void pop ()
  {
    int top, item , stack[10];
     if( top==-1 )
     {
Cout<<"Stack is empty. Underflow condition! ";
     }
     else
     {
        Item=stack[top];
        top = top - 1 ;
     }
  }
```

Linkedlist implementation of Pop operation

- We can pop or delete element from the beginning in linked list stack.
- If no node is present it returns underflow condition.

Operations on stack

1. Push
2. Pop
3. Peek/Top Operation: Peek operation refers to retrieving the topmost element in the stack without removing it from the collections of data elements.
4. isFull() function: isFull function is a function which returns true if stack is full and false otherwise. It is used to check whether or not a stack is empty.
5. isEmpty() function: isEmpty() function is used to check whether or not a stack is empty. It returns true if stack is empty and false otherwise.

Application of Stack Data Structures

1) Expression Evaluation and Conversion(Infix to prefix,Infix to postfix,Prefix to infix, Prefix to postfix, Postfix to infix)
2) Backtracking
3) DFS(Depth First Search)
4) Recursion
5) Parentheses Balancing
6) String Reversal
7) Syntax Parsing
8) Undo/Redo
9) Tower of hanoi

Types of Stacks

1. **Register Stack:** This type of stack is also a memory element present in the memory unit and can handle a small amount of data only. The height of the register stack is always limited as the size of the register stack is very small compared to the memory.
2. **Memory Stack:** This type of stack can handle a large amount of memory data. The height of the memory stack is flexible as it occupies a large amount of memory data.

Space and Time Complexity of Stack

Complexity in Array implementation of stack

OPERATION	BEST TIME COMPLEXITY	WORST TIME COMPLEXITY	AVERAGE TIME COMPLEXITY	SPACE COMPLEXITY
Push	O(1)	O(1)	O(1)	O(1)
Pop	O(1)	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)	O(1)
Search	O(1)	O(n)	O(n)	O(1)

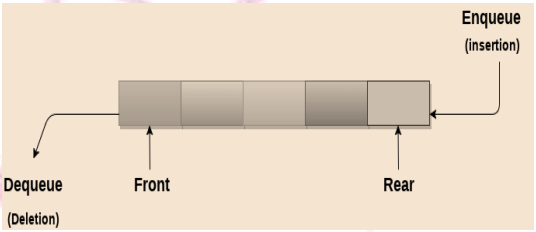
Note:- Here Worst Case Scenario would be O(n) in case of a array implementation of stack where the array is completely filled, then the array size needs to be changed and all the elements must be copied from one array to another , this would result in time being O(n). In case of a LinkedList approach, time remains constant O(1).

Complexity in Linkedlist implementation of stack

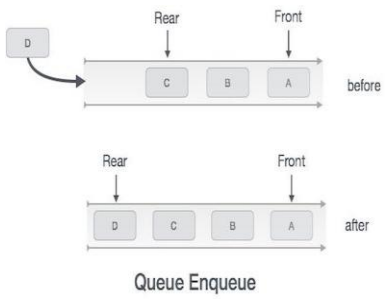
OPERATION	BEST TIME COMPLEXITY	WORST TIME COMPLEXITY	AVERAGE TIME COMPLEXITY	SPACE COMPLEXITY
Push	O(1)	O(N)	O(1)	O(1)
Pop	O(1)	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)	O(1)
Search	O(1)	O(n)	O(n)	O(1)

Queue

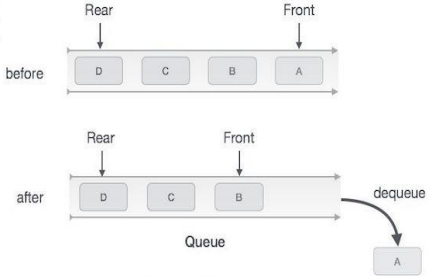
- A queue in C is basically non primitive or abstract linear data structure to store and manipulate the data elements.
- It follows the order of First In First Out (FIFO) or Last in last out(LILO).
- In queues, the first element entered into the array is the first element to be removed from the array.
- Example:- Ticket counter to buy movie tickets, scenario of a bus-ticket booking stall, people waiting in line for a rail ticket.
- Unlike stack, A queue is open at both ends. One end is provided for the insertion of data and the other end for the deletion of data.
- Main difference of the stack and queue is deletion of the item. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



- In a queue insertion operations to be performed at one end called “REAR” and deletion operations to be performed at another end called “FRONT”.
- We can also say that “REAR” points the last element of the queue and “FRONT” points the first element of the queue.
- In a queue insertion operation is called “enqueue” and deletion operation is called “dequeue”.



Queue Enqueue



Queue Dequeue

Operations Associated with a Queue

1. **isEmpty():** To check if the queue is empty
2. **isFull():** To check whether the queue is full or not
3. **dequeue():** Removes the element from the front side of the queue
4. **enqueue():** It inserts elements to the end of the queue
5. **Front:** Pointer element responsible for fetching the first element from the queue
6. **Rear:** Pointer element responsible for fetching the last element from the queue.

We can implement Queue operations 2 ways

a) Array implementation
b) Linkedlist implementation

Working of Queue Data Structure

- Queue follows the First-In-First-Out pattern. The first element is the first to be pulled out from the list of elements.
- Front and Rear pointers keep the record of the first and last element in the queue.
- At first, we need to initialize the queue by setting Front = -1 and Rear = -1
- In order to insert the element (enqueue), we need to check whether the queue is already full i.e. check the condition for Overflow (rear == size – 1).
- If the queue is not full, we'll have to increment the value of the Rear index by 1 and place the element at the position of the Rear pointer variable. When we get to insert the first element in the queue, we need to set the value of Front to 0.
- In order to remove the element (dequeue) from the queue, we need to check whether the queue is already empty i.e. check the condition for Underflow (front == - 1 || front > rear||rear==-1)
- If the queue is not empty, we'll have to remove and return the element at the position of the Front pointer, and then increment the Front index value by 1.
- When we get to remove the last element from the queue, we will have to set the values of the Front and Rear index to -1.

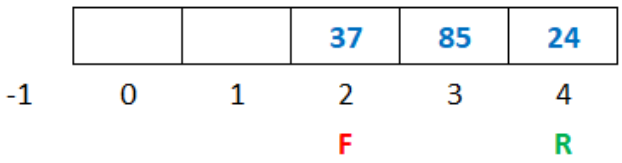
Types of Queue

There are four different types of queue that are listed as follows

1. Simple Queue or Linear Queue
2. Circular Queue
3. Priority Queue
4. Double Ended Queue (or Deque)

Simple queue or Linear Queue

1) In Linear Queue, an insertion takes place from one end while the deletion occurs from another end.
2) The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end.
3) It strictly follows the FIFO rule.
4) The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue.
5) In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.



Complexity of simple Queue of array implementation

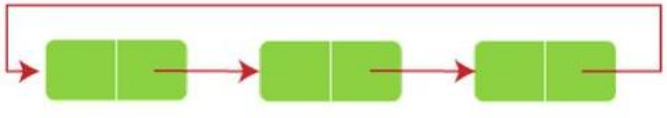
Data Structure	Time Complexity		Space Complexity
Queue	**Average**	**Worst**	**Worst**
Access	θ(n)	O(n)	O(n)
Search	θ(n)	O(n)	
Insertion	θ(1)	O(1)	
Deletion	θ(1)	O(1)	

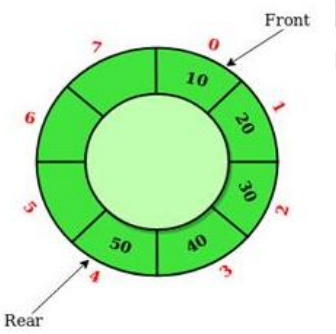
Applications of Queue

1) Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2) Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3) Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4) Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5) Queues are used in operating systems for handling interrupts.
6) Queue are used in traffic light signals.
7) Breadth-First Search Algorithm (BFS).
8) Printer maintains a queue to print document.

Circular Queue

- Circular queue is used to overcome the drawback of linear queue.
- A Circular Queue is a special queue where the queue last element is connected to the first element of the queue forming a circle.
- The operations are performed based on FIFO (First In First Out) principle.
- It is also called 'Ring Buffer'.
- In circular queue, the last node is connected back to the first node to make a circle.
- All the operation of circular queue are same as linear queue only the condition for cheking the queue full is different.
- If queue is full then:-
  (front==0&& rear==size-1)||((rear+1)%size)==front)





Application of circular Queue

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Priority Queue

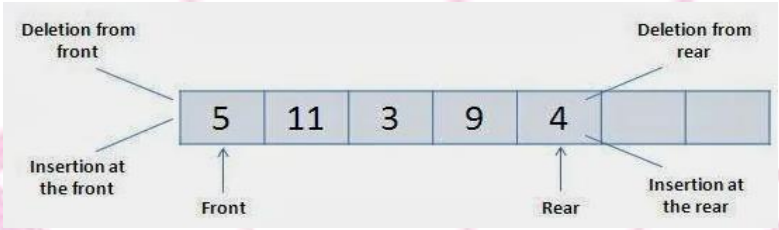
- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.
- In a priority queue, generally, the value of an element is considered for assigning the priority.
- Priority queue does not follow first in first out principle.
- For example:- the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority. The reverse case can also be used.

Application of priority Queue

1. CPU Scheduling
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc.
3. Stack Implementation
4. All queue applications where priority is involved.
5. Data compression in Huffman code
6. Event-driven simulation such as customers waiting in a queue.
7. Finding Kth largest/smallest element.

Deque (or double-ended queue)

- The deque stands for Double Ended Queue.
- Deque is a linear data structure where the insertion and deletion operations are performed from both ends.
- We can say that deque is a generalized version of the queue.
- To implement deque we use circular array or circular doubly linked list.
- Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows:-



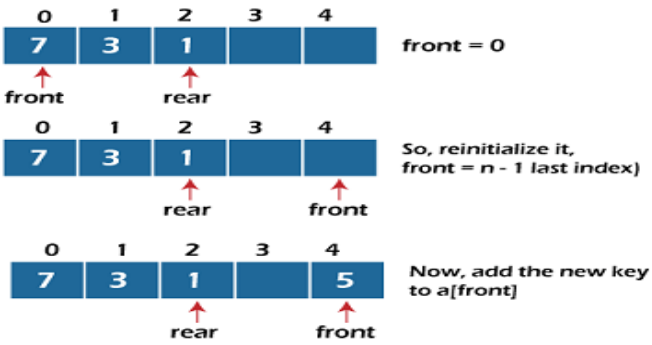
Types of deque

There are two types of deque:

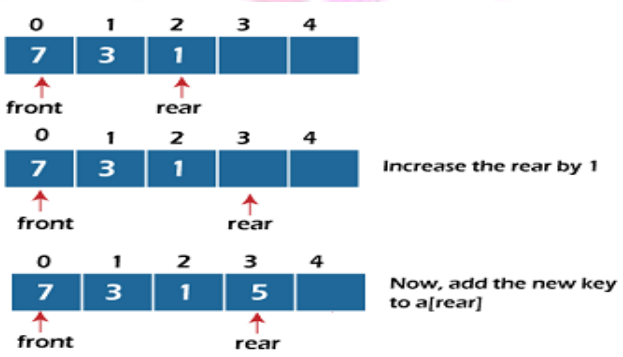
1. **Input restricted queue:-** In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.
2. **Output restricted queue:-** In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

Check empty:- This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.
Check full:- This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

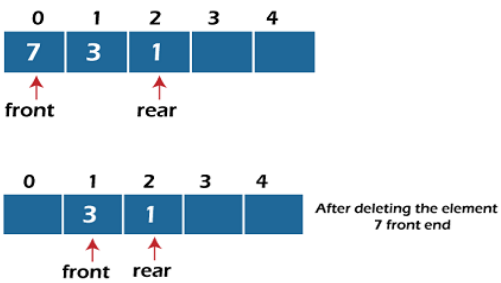
Insertion at the front end (enqueueFront)



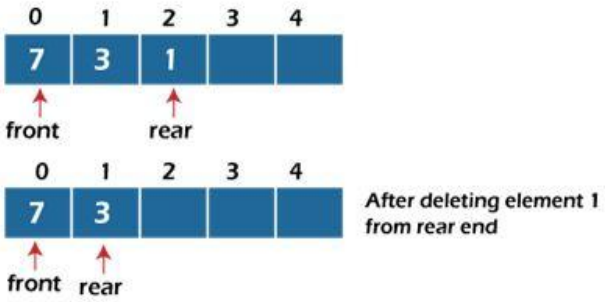
Insertion at the rear end(enqueueRear)



Deletion at the front end(DequeueFront)



Deletion at the rear end(dequeueRear)



Operations on Deque

- **enqueueFront():** Adds an item at the front of Deque.
- **enqueueRear():** Adds an item at the rear of Deque.
- **dequeueFront():** Deletes an item from the front of Deque.
- **dequeueRear():** Deletes an item from the rear of Deque.
- **getFront():** Gets the front item from the queue.
- **getRear():** Gets the last item from queue.
- **isEmpty():** Checks whether Deque is empty or not.
- **isFull():** Checks whether Deque is full or not.

Application on Deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.
- The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.
- Job scheduling algorithm

Stack implementation of queue

We can implement Queue using 2 stacks.

Queue implementation of stack

We can implement stack using 2 queues.

Expression Evaluation
infix notation

- When the operator is written in between the operands, then it is known as infix notation. Operand does not have to be always a constant or a variable; it can also be an expression itself.
- For example:- (p + q) * (r + s)
- In the above expression, both the expressions of the multiplication operator are the operands, i.e., (p + q), and (r + s) are the operands.

Syntax of infix notation
<operand> <operator> <operand>

Operator	*Description*	*Associativity*
() [] . ->>> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to and not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	Comma operator	left to right

Important operators for conversion

Operators	Symbols	Associativity
Parenthesis	( ), {}, [ ]	Left to right
Exponents	^	Right to left
Multiplication and Division	*, /	Left to right
Addition and Subtraction	+ , -	Left to right
BODMAS	We can solve infix operator by BODMAS rule.	

Postfix notation

The Postfix notation is also known as reverse polish notation. The Postfix notation is used to represent algebraic expressions. Postfix expression can be defined as an expression in which all the operators are present after the operands. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

For example:- pq+rs+*

Syntax of postfix notation

<operand> <operand><operator>

AB+

Prefix Notation

The prefix notation is also known as polish notation. The prefix notation is used to represent algebraic expressions. An expression is called the prefix expression if the operator appears in the expression before the operands.

For example:- *+pq+rs

Syntax of prefix notation

<operator><operand> <operand>

+AB

Advantage of Postfix Expression over Infix Expression

- An infix expression is difficult for the machine to know and keep track of precedence of operators.
- On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).
- Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Infix to Postfix conversion

Rules for the conversion from infix to postfix expression

1) Print the operand as they arrive.
2) If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3) If the incoming symbol is '(', push it on to the stack.
4) If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5) If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6) If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7) If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8) At the end of the expression, pop and print all the operators of the stack.

Example:- Infix Expression: A+ (B*C-(D/E)*F)

Scanned	Stack	Postfix Expression	Description
	(		Start with paranthesis
A	(	A	
+	(+	A	
(	(+(	A	
B	(+(	AB	
*	(+(*	AB	
C	(+(*	ABC	
-	(+(-	ABC*	* Is at higher precedence than '–' so pop *from the stack and push it to expression
(	(+(-(	ABC*	
D	(+(-(	ABC*D	
/	(+(-(/	ABC*D	
E	(+(-(/	ABC*DE	
)	(+(-	ABC*DE/	close parenthesis encounters so Pop from top on the stack till open parenthesis encounter
*	(+(-*	ABC*DE/	
F	(+(-*	ABC*DE/F	
)	(+	ABC*DE/F*-	Close parenthesis encounter so pop from top of the stack till open parenthesis encounter
)		ABC*DE/F*-+	Close

Infix to Prefix conversion

Rules for the conversion of infix to prefix expression:

1) First, reverse the infix expression given in the problem.
2) Scan the expression from left to right.
3) Whenever the operands arrive, print them.
4) If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
5) If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
6) If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
7) If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
8) If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
9) When we reach the end of the expression, pop, and print all the operators from the top of the stack.
10) If the operator is ')', then push it into the stack.
11) If the operator is '(', then pop all the operators from the stack till it finds) opening bracket in the stack.
12) If the top of the stack is ')', push the operator on the stack.
13) At the end, reverse the output.

Example:- Infix Expression: A+ (B*C-(D/E)*F),
Reverse the infix expression in the given problem)F*)E/D(-C*B(+A

Scanned	Stack	prefix Expression	Description
	)		Start with paranthesis
	))		
F	))	F	
*	))*	F	
)	))*)	F	
E	))*)	FE	
/	))*)/	FE	
D	))*)/	FED	
(	))*	FED/	open parenthesis encounters so Pop from top on the stack till close parenthesis encounter
-	))	FED/*-	* Is at higher precedence than '-' so pop * from the stack and push it to expression
C	))	FED/*-C	
*	))*	FED/*-C	
B	))*	FED/*-CB	
(	)	FED/*-CB*	open parenthesis encounters so Pop from top on the stack till close parenthesis encounter
+	)+	FED/*-CB*	* Is at higher precedence than '+' so pop * from the stack and push it to expression
A	)+	FED/*-CB*A	
(		FED/*-CB*A+	Close
		+A*BC-*/DEF	Reverse the expression to get prefix notation.

Now reverse the resultant expression to get the prefix expression +A*BC-*/DEF.

Conversion of Postfix to Prefix expression using Stack

The following are the steps used to convert postfix to prefix expression using stack:

1) Scan the postfix expression from left to right.
2) If the element is an operand, then push it into the stack.
3) If the element is an operator, then pop two operands from the stack.
4) Create an expression by concatenating two operands and adding operator before the operands.
5) Push the result back to the stack.
6) Repeat the above steps until we reach the end of the postfix expression. Then we get the resultant prefix expression.

Conversion of Prefix to Postfix Expression using stack

1) Scan the prefix expression from right to left, i.e., reverse.
2) If the incoming symbol is an operand then push it into the stack.
3) If the incoming symbol is an operator then pop two operands from the stack.
4) Once the operands are popped out from the stack, we add the incoming symbol before the operands.
5) When the operator is added before the operands, then the expression is pushed back into the stack.
6) Once the whole expression is scanned, pop, reverse and prints the postfix expression from the stack.

Conversion of Postfix to Infix Expression using stack

Step 1: Create an empty stack used for storing the operands.
Step 2: Scan each element of an expression one by one and do the following:

- If the element is an operand then push it into the stack.
- If the element is an operator then pop two operands from the stack. Perform operation on these operands. Push the final result into the stack.

Step 3: When the expression is scanned completely, the value available in the stack would be the final output of the given expression.

Example:-
If the expression is: 5, 6, 2, +, *, 12, 4, /, -

Scanned	Stack	Infix	Infix evaluation
5		5	5
6		5,6	5,6
2		5,6,2	5,6,2
+	+	5,6+2	5,8
*	*	5*(6+2)	40
12		5*(6+2),12	40,12
4		5*(6+2),12,4	40,12,4
/	/	5*(6+2), (12/4)	40,3
-	-	5*(6+2)-(12/4)	37

Conversion of prefix to Infix Expression using stack

Algorithm for the evaluation of postfix expression using stack:
Step 1: Create an empty stack used for storing the operands.
Step 2: Scan each element of an expression one be one and do the following:

- If the element is an operand then push it into the stack
- If the element is an operator then pop two operands from the stack. Reverse it and perform operation on these operands. Push the final result into the stack.

Step 3: When the expression is scanned completely, the value available in the stack would be the final output of the given expression.

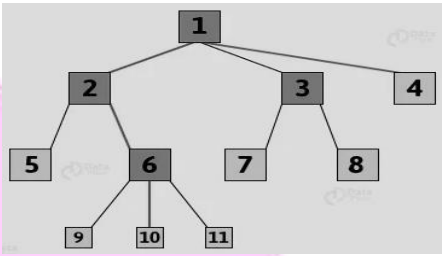
Example:- Expression: +, -, *, 2, 2, /, 16, 8, 5
First, we will reverse the expression given above.
Expression: 5, 8, 16, /, 2, 2, *, -, +

Scanned	Stack	Infix	Infix evaluation
5		5	5
8		5,8	5,8
16		5,8,16	5,8,16
/	/	5,16/8	5,2
2		5,16/8,2	5,2,2
2		5,16/8,2,2	5,2,2,2
*	*	5,16/8,2*2	5,2,4
-	-	5,(2*2)-(16/8)	5,2
+	+	((2*2)-(16/8))+5	7

Complexity

Space and time complexity of expression evaluation by using stack is O(n)

Tree Data Structure



- In programming terminology, a tree is nothing but a non-linear data structure that has multiple nodes. Tree represents the nodes connected by edges. Trees are hierarchical data structures.
- Traversing in a tree is done by depth first search and breadth first search algorithm.
- It has no loop and no circuit.
- It has no self-loop. It is hierarchical model.

Types of Tree data structures

The different types of tree data structures are as follows:

1. **General tree:-** A general tree data structure has no restriction on the number of nodes. It means that a parent node can have any number of child nodes.
2. **Binary tree:-** A node of a binary tree can have a maximum of two child nodes. In the given tree diagram, node B, D, and F are left children, while E, C, and G are the right children.
3. **Binary search tree:-** As the name implies, binary search trees are used for various searching and sorting algorithms. The examples include AVL tree and red-black tree. It is a non-linear data structure. It shows that the value of the left node is less than its parent, while the value of the right node is greater than its parent.
4. **Balanced tree:-** If the height of the left sub-tree and the right sub-tree is equal or differs at most by 1, the tree is known as a balanced tree.

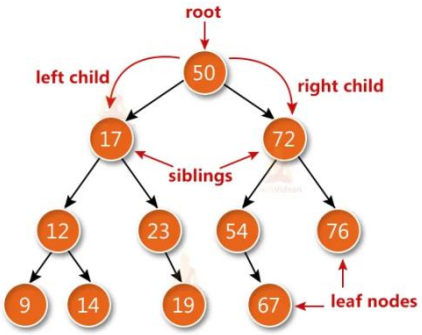
Applications of Tree data structure

The applications of tree data structures are as follows:

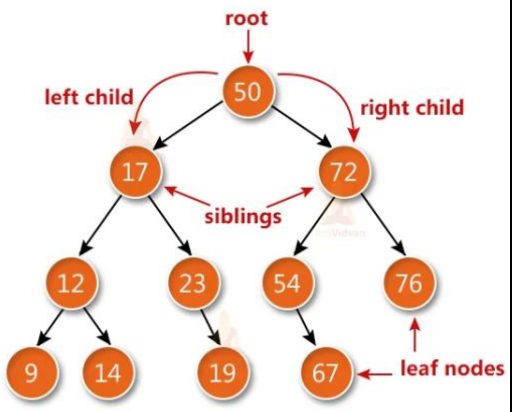
1. **Spanning trees:** It is the shortest path tree used in the routers to direct the packets to the destination.
2. **Storing hierarchical data:** Tree data structures are used to store the hierarchical data, which means data is arranged in the form of order.
3. **Syntax tree:** The syntax tree represents the structure of the program's source code, which is used in compilers.
4. **Heap:** It is also a tree data structure that can be represented in a form of an array. It is used to implement priority queues.
5. **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.
6. Using tree can form multi stage decision making
7. Huffman coding trees are used in data compression algorithms.

Binary tree

- A binary tree is the special version of the General tree where each node has atmost 2 child.
- Every node in a binary tree has a left and right reference along with the data element.
- The node at the top of the hierarchy of a tree is called the root node.
- Time and space complexity of Binary Tree is O(n).
- A binary tree is a tree data structure whose all nodes have zero, one, or at most two children nodes. These two children are generally referred to as left and right child respectively.
- The top-most node is known as the root node, while the nodes with no children are known as leaf nodes.



Some important terms related to tree



1. **Nodes:-**
2. **Root:-** The topmost node
3. **Parent Node:-** a node which has child node is called parent nodes.
4. **Child Node:-** A node that has a preceding node is known as a child node.
5. **Siblings:-** Siblings mean that nodes which have the same parent node.
6. **External node or Leaf Node:-** A node with no children.
7. **Internal Node:-** A node that has at least one child node is known as an internal node.
8. **Ancestors or predecessors:-** Ancestors include the parent, grandparent and so on of a node.
9. **Descendants or successor:-** Descendants includes the child, grandchild and so on of a node.
10. **Edge:-** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges.
11. **Path:-** Path is a combination of nodes and edges connected with each other.
12. **Degree of node:-** Degree of a node implies the number of child nodes a node has. Degree of leaf node is always 0.
13. **Depth of a node:-** The depth of a node is defined as the length of the path from the root to that node.
14. **Height of a node:-** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
15. **Level of a node:-** Level of a node is equal to depth of the node.
16. **Degree of a tree:-** The degree of a tree is the maximum degree of a node among all the nodes in the tree. In the binary tree degree of tree is always 2.
17. **Depth of the binary tree:-** The number of edges from a deepest node in the tree to the root node.
18. **Height of a binary tree:-** The number of edges from the deepest node in the tree to the root node.
19. **Level of a binary tree:-** Level of a tree is equal to height of the tree.
    Note:- height of tree = level of tree= depth of tree. But depth of node = level of node but may or may not be equal to height of node.

Properties of Binary Tree

- **Property 1:-** In a binary tree At each level of L or height h, the maximum number of nodes is 2^L or 2^h.
- **Property 2:-** The maximum number of nodes n possible at height h is = 2^{h+1} -1.
- **Property3:-** The minimum number of nodes n possible at height h is equal to h+1 (n=h+1).
- **Property 4:-** In a Binary Tree with n nodes, minimum possible height or the minimum number of levels is h = $\log_2(n+1) - 1$.
- **Property 5:-** In a Binary Tree with n nodes, maximum possible height or the maximum number of levels is: h = n-1.
- **Property 6:-** In Binary tree where every node T has 0 or 2 children, the number of leaf nodes l is always one more than nodes with two children. l=T+1
- **Property 7:-** In a non empty binary tree, if n is the total number of nodes and e is the total number of edges, then e = n-1.
- **Property 8:-** A Binary Tree with l leaves has at least | $\text{Log}_2 l$ | levels.

Types of Binary Tree

There are four types of Binary tree:

1. Full/ proper/ strict Binary tree
2. Complete Binary tree
3. Perfect Binary tree
4. Degenerate Binary tree
5. Balanced Binary tree

Implementation of binary tree using array(Sequential Representation)

- An array can be converted into a binary tree. To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n + 1.
- **Case1:** when array index starts from 0. Consider an array a[10]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
A	B	C	D	E	F	G	H	I	J

If a node is at i^{th} index

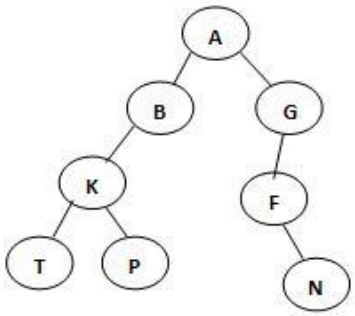
a. **Parent :** Parent would be at floor((i-1)/2) except the root.
b. **Left Child :** Left child would be at 2*i+1.
c. **Right Child :** Right child would be at 2*i +2.
d. **Left Sibling :** left Sibling of a node at index i lies at (i-1).
e. **Right Sibling :** Right sibling of a node at index i lies at (i+1).

Basic Operation On Binary Tree

There are some following operation on Binary tree:-

1. **Searching:** Worst case complexity of binary tree searching is O(n).
2. **Insertion:** Worst case complexity of binary tree insertion is O(n).
3. **Deletion:** Worst case complexity of binary tree deletion is O(n).
4. **Traversing:** 2 traversal techniques for a binary tree that are:

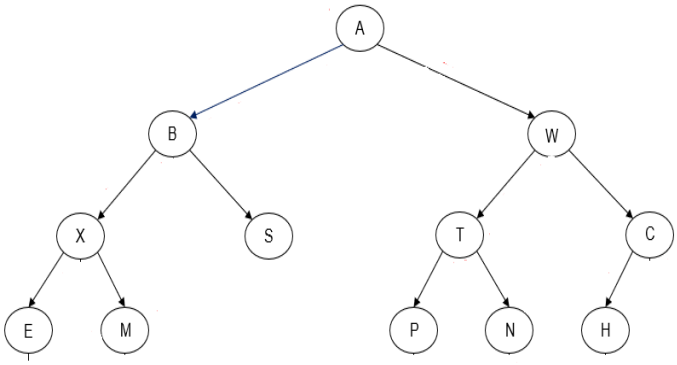
a) **Breadth First search or level order traversal:-** Breadth-first search is a tree traversal which starts traversing the tree from the root node and explores all the neighboring nodes at the same level.
It is also called level order traversing. BFS uses Queue data structure to perform operations.
Example:- Level order traversing of the above given example: ABGKFTPN



b) **Depth First search:-** Depth First search is implemented by using stack data structure.
a) **Preorder traversal:**

- Visit the root.
- Traverse the left sub tree of root recursively.
- Traverse the right sub tree of root recursively.
- Note: preorder traversal is also known as Root Left Right traversal.

Preorder of above tree:- ABXEMSWTPNCH



b) Inorder traversal

- Traverse the left most sub tree recursively.
- Visit the root.
- Traverse the right most sub tree recursively.
- Note: Inorder traversal is also known as Left Root Right traversal.

Inorder of above tree:- EXMBSAPTNWHC

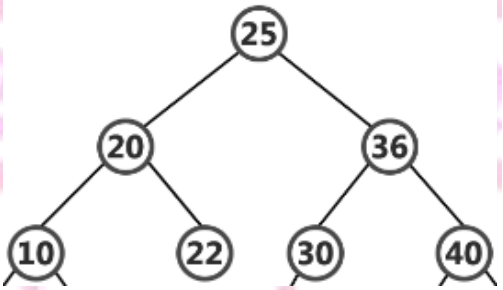
c) Postorder traversal

- Traverse the left sub tree of root recursively.
- Traverse the right sub tree of root recursively.
- Visit the root.
- Note: Postorder traversal is also known as Left Right Root traversal.

postorder of above tree:- EMXSBPNTHCWA

Binary Search Tree

- Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.
- Binary search tree is a variation of binary tree which is specially made for searching. BST is an ordered data structure.
- A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –
  - ✓ The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
  - ✓ The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.
  - ✓ The left and right subtree each must also be a binary search tree.
  - ✓ There must be no duplicate nodes.
  - ✓ Time complexity of insertion, deletion, searching and traversing is depends on the height of binary tree.
  - ✓ In average case height of binary tree O(h)= O(logn) and in worst case O(h)=O(n).

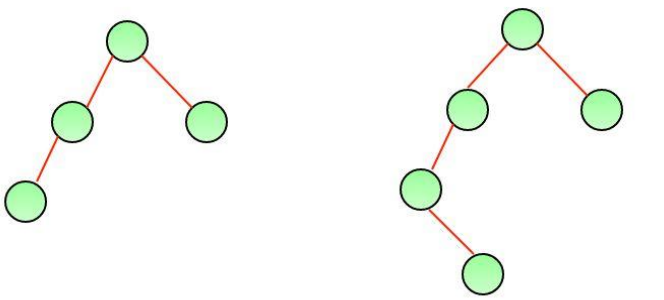


Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Balanced tree

- A balanced binary tree is also known as height balanced tree. It is defined as binary tree in when the difference between the height of the left subtree and right subtree is not more than 1,
- An empty tree is height-balanced. A non-empty binary tree T is balanced if:
  - ✓ Left subtree of T is balanced
  - ✓ Right subtree of T is balanced
  - ✓ The difference between heights of left subtree and the right subtree is not more than 1.
- Time and space complexity of balanced binary search tree is O(n).



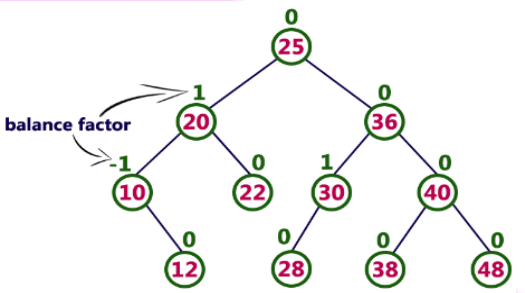
A height balanced tree Not a height balanced tree

AVL Tree

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- AVL tree is a perfectly balanced tree.
- This difference is called the Balance Factor.
- In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor = height Of Left Subtree – height Of Right Subtree

- Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.
- If the tree is not balance we do some rotation to balance the tree.

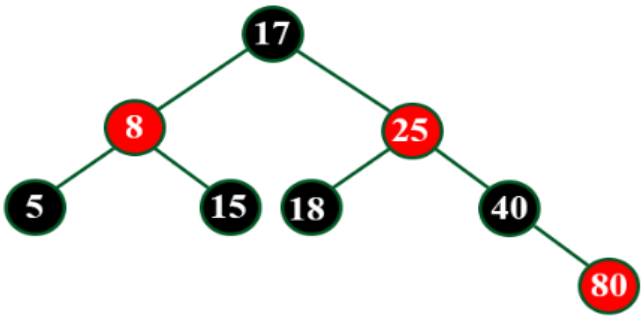


AVL Tree Rotations

- In AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.
- Rotation is the process of moving nodes either to left or to right to make the tree balanced.
- An AVL tree may perform the following four kinds of rotations:-
  - **a)** Left rotation
  - **b)** Right rotation
  - **c)** Left-Right rotation
  - **d)** Right-Left rotation

Red Black tree

- A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black).
- These colors are used to ensure that the tree remains balanced during insertions and deletions.
- Although the balance of the tree is not perfect, but it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.
- This tree was invented in 1972 by Rudolf Bayer.
- Red-black tree takes less time to structure the tree by restoring the height of the binary tree.



Properties of Red Black Tree

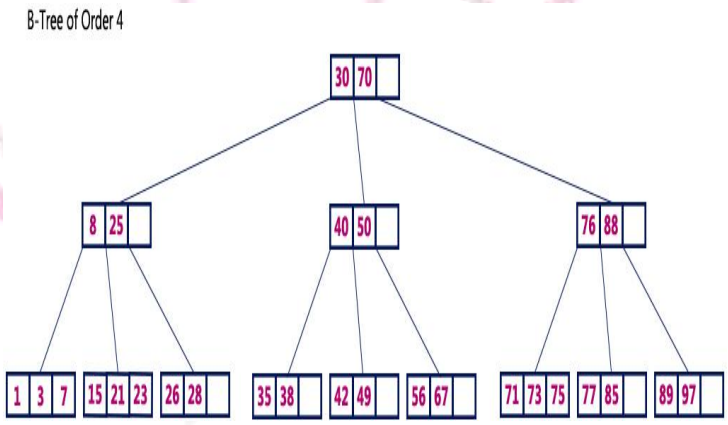
Property #1: Red - Black Tree must be a Binary Search Tree.
Property #2: The ROOT node must be colored BLACK.
Property #3: The children of Red colored node must be colored BLACK. (There should not be two adjecent RED nodes).
Property #4: In all the paths of the tree, there should be same number of BLACK colored nodes.
Property #5: Every new node must be inserted with RED color.
Property #6: Every leaf (i.e NULL node) must be colored BLACK.

Important points about Red-Black Tree

- The black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes (nil) are also counted as black nodes.
- Red-black tree of height h has black height >= h/2.
- Height of a red-black tree with n nodes is h<= 2 log2(n + 1).
- All leaves (NIL) are black.
- The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.

B tree

- B Tree is a special type of search tree in which a node contains more than one value (key) and more than two children.
- B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.
- B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.
- The number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.
- B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size.
- As we know every process is in main memory and disk access time is very high compared to the main memory access time.
- If there is a huge amount of data that cannot fit in main memory. When the number of keys are high, the data is read from disk in the form of blocks.
- The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree.

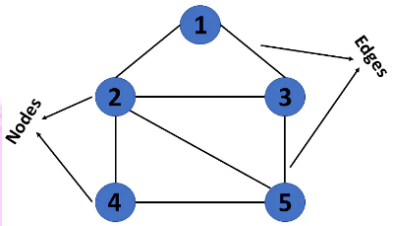


B-Tree of Order m has the following properties

Property #1:- All leaf nodes must be at same level.
Property #2:- All nodes except root must have atleast or minimum [m/2]-1 keys and maximum of m-1 keys.
Property #3:- All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
Property #4:- If the root node is a non leaf node, then it must have atleast 2 children.
Property #5:- A non leaf node with n-1 keys must have n number of children.
Property #6:- All the key values in a node must be in Ascending Order.

Graph DataStructure

- A Graph is a non-linear data structure consisting of vertices and edges. The vertices are also referred to as vertices and the edges are lines that connect any two vertices in the graph.
- In formal way Graph G(V, E) can be defined as set of vertices(or nodes) V and a set of Edges E that connect a pair of vertices.
- Example:-
  This graph has a set of vertices V= { 1,2,3,4,5} and a set of edges
  E= { (1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(4,5}.
- Graphs are used to represent networks.
- The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like linkedIn, Facebook.

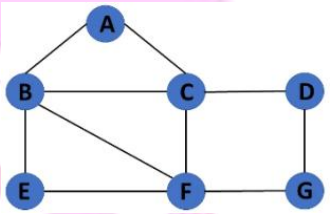


Types of Graphs in Data Structures

There are following different types of graphs in data structures:-

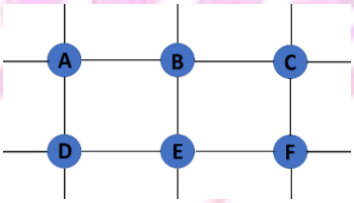
1) Finite Graph:-

The graph G=(V, E) is called a finite graph if there is limited number of vertices and edges in the graph.



2) Infinite Graph:-

The graph G=(V, E) is called a finite graph if there are infinite number of vertices and edges in the graph.

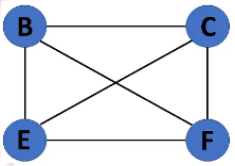


3) Trivial Graph:-

A graph G= (V, E) is trivial if it has only a single vertex and no edges.

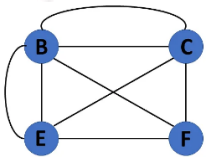
4) Simple Graph:-

A graph G=(V, E) is a simple graph If each pair of vertices has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices.



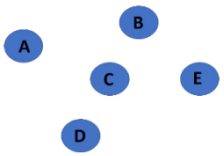
5) Multi Graph:-

A graph G=(V, E) is a multi graph If there are parallel edges between a pair of vertices There are no self-loops in a Multigraph.



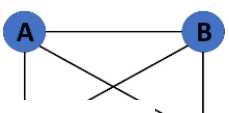
6) Null Graph:-

A graph G= (V, E) is a null graph If there are several vertices but no edge to connect them. It's a revised version of a trivial graph.

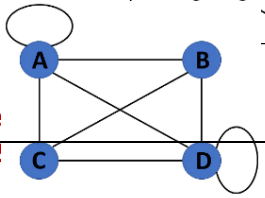


7) Complete Graph:-

A complete graph G= (V, E) is also a simple graph. Each vertex is connected with other n vertex. It's also known as a full graph because each vertex's degree must be n-1.



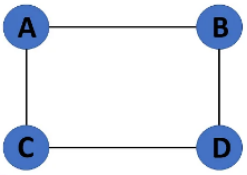
8) Pseudo Graph:-



A graph G= (V, E) is a pseudograph if it contains a self-loop besides other edges.

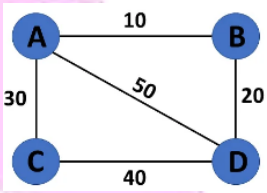
9) Regular Graph:-

A graph G= (V, E) is a regular graph or simple graph If the graph has same degree at each vertex. As a result, every whole graph is a regular graph.



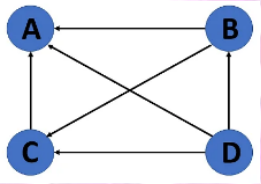
10) Weighted Graph:-

A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



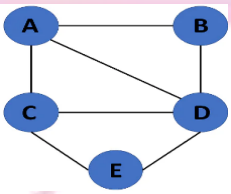
11) Directed Graph:-

A directed graph is a graph where each edge has a direction. It is also referred to as a digraph.



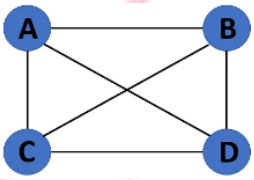
12) Undirected Graph:-

An undirected graph is a set of vertices and links connecting them. Two connected vertices have no direction. You can form an undirected graph with a finite number of vertices and edges.



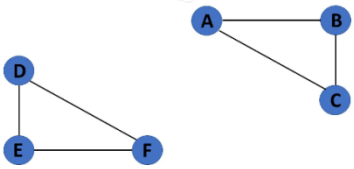
13) Connected Graph:-

If there is a path between one vertex to any other vertex then the graph is called connected.



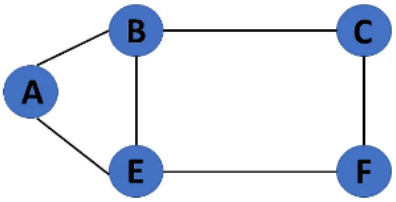
14) Disconnected Graph:-

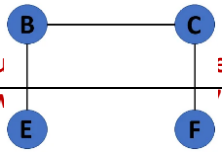
If there is no edge connecting the vertices, is called disconnected graph. We refer to the null graph as a disconnected graph.



15) Cyclic Graph:-

If a graph contains at least one graph cycle, it is called cyclic graph.



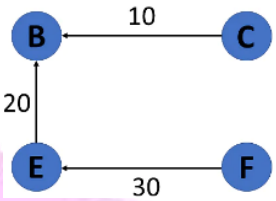


16) Acyclic Graph:-

If there are no cycles in a graph, it is called an acyclic graph.

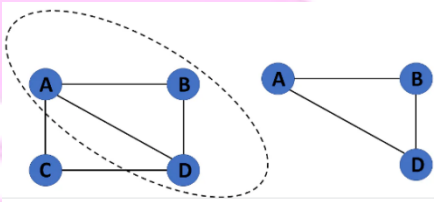
17) Directed Acyclic Graph:-

It's also known as a Directed Acyclic Graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



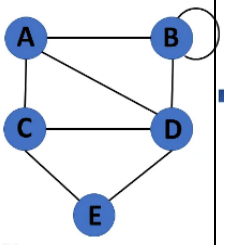
18) Subgraph:-

The vertices and edges of a graph that are subsets of another graph are known as a subgraph.



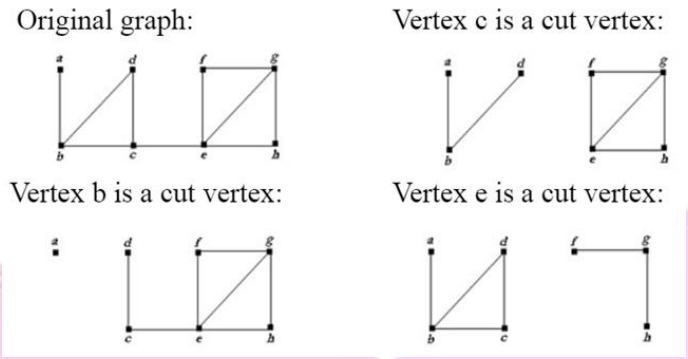
Terminologies of Graphs in Data Structures

- **Vertex or node:-** Each endpoint of a line or edge in the graph is called as a vertex or node.
- **Edge:-** To connect 2 vertices we use edge or line. An edge has two ends called vertices. Edge represents a path between two vertices
- **Adjacent vertex:-** If two vertices are endpoints of the same edge, they are adjacent vertex.
- **Source vertex:-** A vertex with an in-degree of zero is referred to as a source vertex,
- **Destination or sink vertex:-** A vertex with an out-degree of zero is known as destination or sink vertex.
- **Isolated vertex:-** An isolated vertex is a zero-degree vertex that is not an edge's endpoint.
- **Adjacent edge:-** If 2 or more edges are connected with same vertex then these edges are called adjacent edge
- **Outgoing edge:-** A vertex's outgoing edges are directed edges that point to the origin.
- **Incoming edge:-** A vertex's incoming edges are directed edges that point to the vertex's destination.
- **Parallel edges:-** When multiple edges in a graph connect the same pair of vertices or nodes, the edges are referred to as parallel edges.
- **Path:-** Path represents a sequence of edges between the two vertices. A path with unique vertices is called a simple path.
- **Degree of vertex:-**The total number of edges connected to a vertex in a graph is called its degree.
- **Out degree:-** The out-degree of a vertex in a directed graph is the total number of outgoing edges.
- **In degree:-** The in-degree of a vertex in a directed graph is the total number of incoming edges.
- **Cycle:-**The path that starts and finishes at the same vertex is known as a cycle.
- **Bridge:-** A bridge is an edge,removal of that edge disconnect the graph.
- **Digraph:-** A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

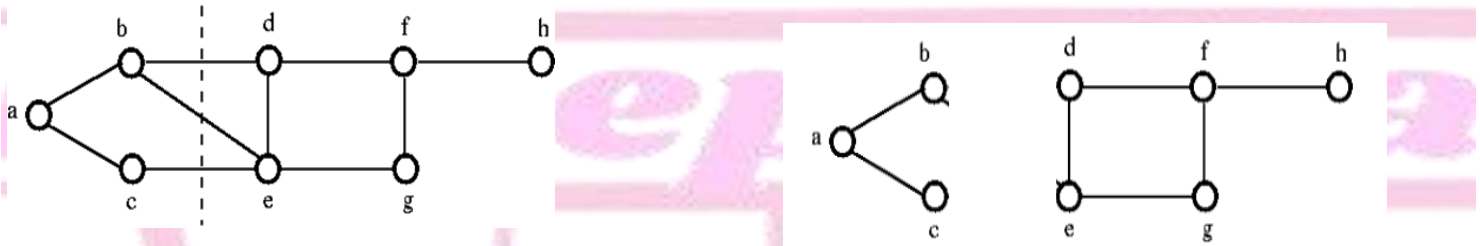


- **Cut Vertex:** A single vertex whose removal disconnects a graph is called a cut-vertex.

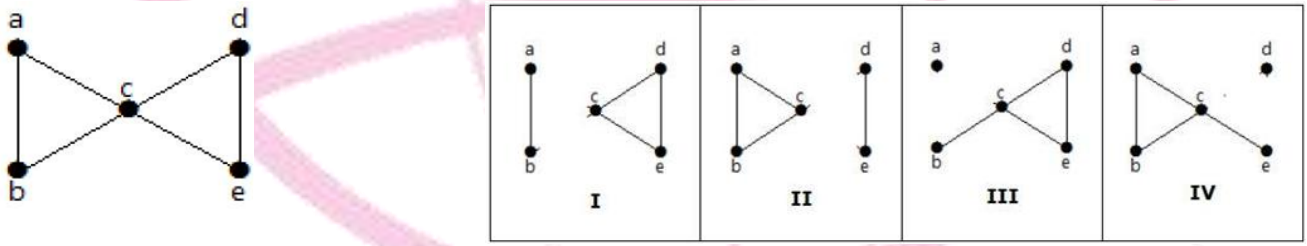
- **Cut Edge (Bridge):** A cut- Edge or bridge is a single edge whose removal disconnects a graph.



- **Cut Set:**In a connected graph G, a cut set is a set S of edges with the following properties:
  - The removal of all the edges in S disconnects G.
  - The removal of some of edges (but not all) in S does not disconnect G.

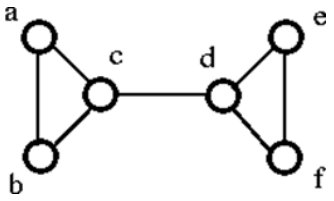


- **Edge Connectivity:** The edge connectivity of a connected graph G is the minimum number of edges whose removal makes G disconnected. It is denoted by λ(G).
  When λ(G) ≥ k, then graph G is said to be k-edge-connected.



- **Vertex Connectivity:** The connectivity (or vertex connectivity) of a connected graph G is the minimum number of vertices whose removal makes G disconnects or reduces to a trivial graph. It is denoted by K(G).
  The graph is said to be k- connected or k-vertex connected when K(G) ≥ k. To remove a vertex we must also remove the edges incident to it.

  Example:-
  The graph G can be disconnected by removal of the single vertex either 'c' or 'd'. Hence, its vertex connectivity is 1. Therefore, it is a 1-connected graph.



- For each pair of vertices x, y, a graph is strongly connected if it contains a directed path from x to y and a directed path from y to x.

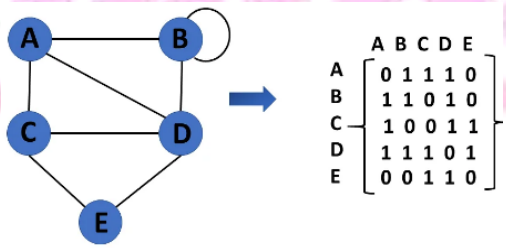
- A directed graph is weakly connected if all of its directed edges are replaced with undirected edges, resulting in a connected graph. A weakly linked graph's vertices have at least one out-degree or in-degree.
- A tree is a connected forest. The primary form of the tree is called a rooted tree, which is a free tree.
- A spanning subgraph that is also a tree is known as a spanning tree.
- Forest is a graph without a cycle.
- Total number of edges in a complete graph:
  a. **If graph is undirected then total number of edges**
     $^{n}C_2 = n(n-1)/2$
  b. **If graph is directed then total number of edges**
     $2 * {}^{n}C_2 = n(n-1)$

Representation of Graphs in Data Structures

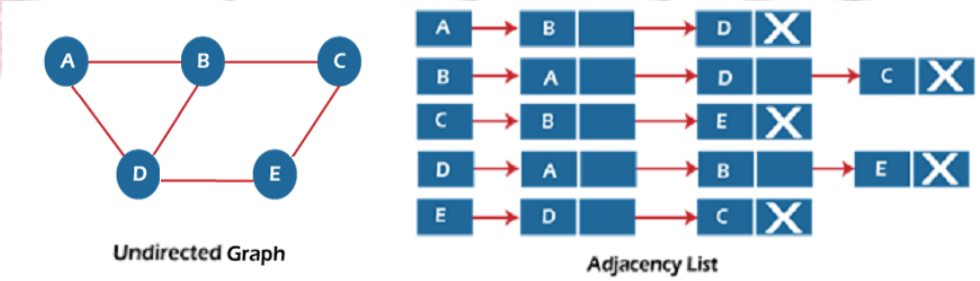
Graphs in data structures are used to represent the relationships between objects. Every graph consists of a set of points known as vertices or nodes connected by lines known as edges.
There are two ways to store Graphs into the computer's memory:

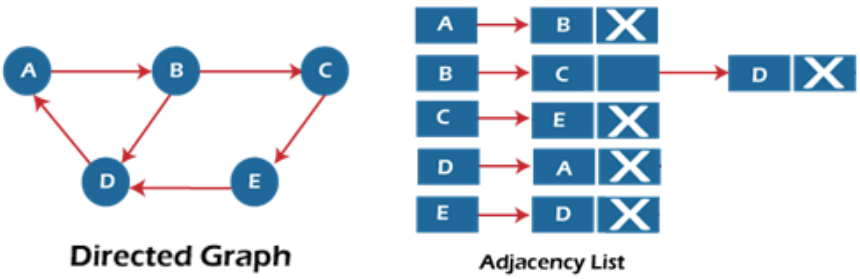
a) **Adjacency matrix**
b) **Adjacency list**

Adjacency matrix:-



Adjacency List:-





Operations on Graphs in Data Structures

a) Creating graphs
b) Insert vertex

c) Delete vertex
d) Insert edge
e) Delete edge
f) Traversing

Applications of Graph

- GPS systems and Google Maps use graphs to find the shortest path from one destination to another.
- The Google Search algorithm uses graphs to determine the relevance of search results.
- World Wide Web is the biggest graph.
- Social Networks like facebook, twitter, etc. use graphs to represent connections between users.
- The nodes we represent in our graphs can be considered as the buildings, people, group, landmarks or anything in general , whereas the edges are the paths connecting them.

Spanning Tree

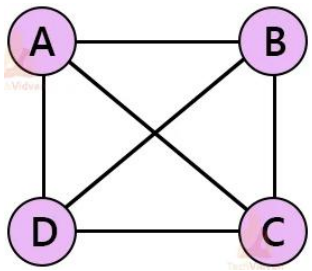
- A spanning tree is a tree that connects all the vertices of a graph with the minimum possible number of edges. Thus, a spanning tree is always connected.
- A spanning tree never contains a cycle.
- A spanning tree is always defined for a graph and it is always a subset of that graph. Thus, a disconnected graph can never have a spanning tree.
- Every undirected and connected graph has a minimum of one spanning tree.
- Consider a graph having V vertices and E number of edges.
  Then, we will represent the graph as G(V, E).
  Its spanning tree will be represented as G'(V, E')
  where $E' \subseteq E$
  V=V' (the number of vertices remain the same).
- A spanning tree G' is a subgraph of G whose vertex set is the same but edges may be different.

Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is minimally connected, so removing one edge from the tree will make the graph disconnected.
- A spanning tree is maximally acyclic, so adding one edge to the tree will create a loop.
- There can be a maximum n^{n-2} number of spanning trees that can be created from a complete graph. where 'n' is the number of nodes.
- A spanning tree has n-1 edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.
- So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Example:- Here we have a complete graph G(n,e)= G(4,6).
Where v is number of vertices and E is number of edges.
n=4 and e=6
Maximum number of spanning tree:- $n^{n-2} = 4^{4-2} = 4^2 = 16$.
Remove e-n+1 edges to create spanning tree for complete graph 6-4+1 =3
There are n-1 edges are there in spanning tree n-1= 4-1 =3

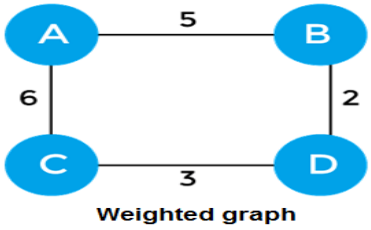


Minimum spanning tree

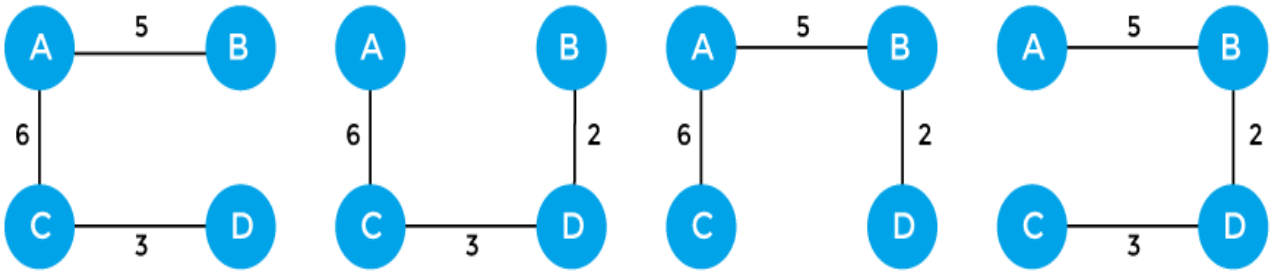
- A minimum spanning tree is defined for a weighted graph. In minimum spanning tree the sum of the weights of the edges is minimum.

- A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree.
- In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

Example of minimum spanning tree:-



Weighted graph



Sum = 14
Minimum spanning tree - 1

Sum = 11
Minimum spanning tree - 2

Sum = 13
Minimum spanning tree - 3

Sum = 10
Minimum spanning tree - 4

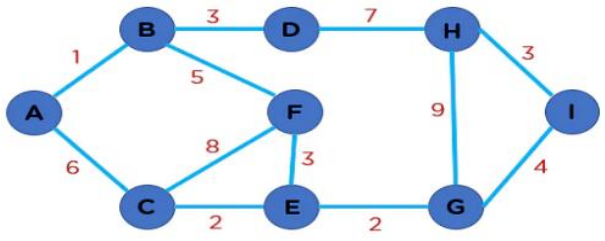
Minimum Spanning Tree Algorithms

There are various algorithms in computer science that help us find the minimum spanning tree for a weighted graph. Some of these algorithms are:

1. Prim's Algorithm
2. Kruskal's Algorithm

1. Prim's Algorithm

- Prim's algorithm uses Greedy approach to find the minimum spanning tree.
- Prim's algorithm always starts with a single vertex and it moves through several adjacent vertices, in order to explore all of the connected edges along the way.
- Prims algorithm starts with adjacent vertices of a source vertex to make a locally optimal choice, to find the global optimal solution.
- Prim's algorithm only create a minimum spanning tree of connected graph it is not used for disconnected graph.
- Example:- Graph G(V, E) given below contains 9 vertices and 12 edges. We are supposed to create a minimum spanning tree T(V', E') for G(V, E) such that the number of vertices in T will be 9 and edges will be 8 (9-1).



Graph G(V, E)

Complexity of Prim's algorithm

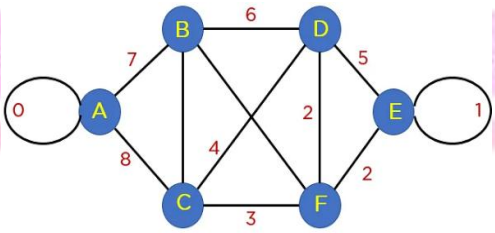
The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. If E is the no. of edges, and V is the no. of vertices.

Data structure used for the minimum edge weight	Time Complexity	Space complexity
Adjacency matrix, linear searching	$O(V^2)$	O(1)
Adjacency list and binary heap	O(E logV) or O(V logV)	O(1)
Adjacency list and Fibonacci heap	O(E+ V logV)	O(1)

Kruskal's Algorithm

- Kruskal's algorithm uses the greedy approach to find the minimum cost spanning tree.
- In Kruskals algorithm we choose the smallest weight edge that does not cause a cycle in the MST.
- Kruskal's algorithm sorts all the edges in increasing order of their edge weights and keeps adding nodes to the tree only if the chosen edge does not form any cycle.
- It picks the edge with a minimum cost at first and the edge with a maximum cost at last. Hence, we can say that the Kruskal algorithm makes a locally optimal choice, intending to find the global optimal solution. That is why it is called a Greedy Algorithm.
- We can also implement Kruskal algorithms using the Union Find Algorithm with optimal time complexity O(ElogV).
- Unlike Prim's algorithm we can apply Kruskal's algorithm in both connected and disconnected graphs.

Example:- The graph G(V, E) given below contains 6 vertices and 12 edges. And you will create a minimum spanning tree T(V', E') for G(V, E) such that the number of vertices in T will be 6 and edges will be 5 (6-1).



Graph G(V, E)

Complexity of Kruskal's algorithm

- If E is the no. of edges, and V is the no. of vertices then the running time of the Kruskal's algorithm by using the above example is $O(V^2)$.
- We can optimize this complexity to O(ElogV) or O(ElogE) by using Union find algorithm to implement Kruskal's Algorithm. As we know for complete graph E is approximately equal to V^2 so O(logV) and O(logE) are same.
- Space complexity is O(V+E).

Shortest path problem

Shortest path problem is a problem of finding the shortest path(s) between vertices of a given graph.

Types of Shortest path problem

There are some following types of shortest path problem:-

1. Single-pair shortest path problem
2. Single-source shortest path problem
3. Single-destination shortest path problem
4. All pairs shortest path problem

1. Single-Pair Shortest Path Problem

- It is a shortest path problem where the shortest path between a given pair of vertices is computed.

- A* Search Algorithm is a famous algorithm used for solving single-pair shortest path problem.

2. Single-Source Shortest Path Problem

- It is a shortest path problem where the shortest path from a given source vertex to all other remaining vertices is computed.
- Dijkstra's Algorithm and Bellman Ford Algorithm are the famous algorithms used for solving single-source shortest path problem.

3. Single-Destination Shortest Path Problem

- It is a shortest path problem where the shortest path from all the vertices to a single destination vertex is computed.
- By reversing the direction of each edge in the graph, this problem reduces to single-source shortest path problem.
- Dijkstra's Algorithm is a famous algorithm adapted for solving single-destination shortest path problem.

4. All Pairs Shortest Path Problem

- It is a shortest path problem where the shortest path between every pair of vertices is computed.
- Floyd-Warshall Algorithm and Johnson's Algorithm are the famous algorithms used for solving All pairs shortest path problem.

Dijkstra's algorithm

- Dijkastra algorithm is essentially a weighted version of breadth-first search: BFS uses a FIFO queue; while Dijkstra's algorithm uses a priority queue.
- Dijkastra algorithm uses greedy approach to find the shortest path.
- Dijkastra algorithm may or may not work for negative-weight edges. But it is completely failed for negative weight edge cycle.
- Dijkstra's algorithm is very similar to Prim's algorithm(for minimum spanning tree) with complexity $O(V^2)$. Like Prim's MST, generate a SPT (shortest path tree) with a given source as a root.
- Dijkstra algorithm works only for connected graphs.
- The actual Dijkstra algorithm does not output the shortest paths. It only provides the value or cost of the shortest paths.
- Dijkstra algorithm works for directed as well as undirected graphs.

Complexity of Dijkstra Algorithm

- If we are using adjacency matrix like Prim's algorithm then the Time complexity of Dijkstra is $O(V^2)$.
- If we use heap (or priority queue) then the optimized time complexity of Dijkstra algorithm is O(ElogV).

Bellman Ford Algorithm

- Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph.
- Bellman Ford Algorithm is based on the Dynamic Programming approach.
- If the weighted graph contains the negative weight edges, the Dijkstra algorithm does not confirm whether it produces the correct answer or not.
- To overcome this problem of Dijkstra algorithm we use bellman ford algorithm that guarantees the correct answer even if the weighted graph contains the negative weight edges
- Bellman-Ford does not work with an undirected graph with negative edges as it will be declared as a negative cycle.

Rule of Bellman Ford algorithm:

We will go on relaxing all the edges (n - 1) times where, n is number of vertices.

Complexity Analysis of Bellman Ford

Time Complexity:-

Since we are traversing all the edges V-1 times, and each time we are traversing all the vertices, therefore the time complexity is O(V.E).

Space Complexity:-

Since we are using an auxiliary array d[] of size V, the space complexity is O(V).

Where V and E are numbers of vertices and edges respectively.

All pair shortest path

- As the name suggest in this method we can find the shortest path from each vertex v to every other u.
- However storing all the paths explicitly needs one spanning tree for each vertex which could consume very much memory and also expensive.
- This is often impractical regarding memory consumption, therefore, these are generally considered as all pairs-shortest distance problems, which aim to find just the distance from each node to another.
- To get the output we use matrix where the entry in u's row and v's column should be the weight or distance of the shortest path from u to v.
- In single-source algorithms, we use an adjacency list representation of the graph, and in all pair shortest path we use an adjacency matrix representation.
- We find the all pair shortest path by using Floyd Warshall's Algorithm.

Floyd Warshall's Algorithm

- Floyd Warshall algorithm uses dynamic programming approach to find the all pair shortest path.
- Let the vertices of graph G be V = {1, 2.....n} and consider a subset {1, 2.....k} of vertices for some k.
- For any pair of vertices i, j ∈ V, considered all paths from i to j whose intermediate vertices are all drawn from {1, 2.....k}, and let p be a minimum weight path from amongst them.
- The Floyd-Warshall algorithm exploits a link between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2.....k-1}. The link depends on whether or not k is an intermediate vertex of path p.
- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2.....k-1}. Thus, the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k-1} is also the shortest path i to j with all intermediate vertices in the set {1, 2.....k}.
- If k is an intermediate vertex of path p, then we break p down into i → k → j.
- Let $d_{ij}(k)$ be the weight of the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k}.

  A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Time Complexity of Floyd Warshall's Algorithm

The running time of the Floyd-Warshall algorithm is determined by the triple nested for loops.

The time complexity of algorithm is $O(n^3)$.

The space complexity of algorithm is$O(n^2)$.

Searching

Searching is a process of finding a particular element among several given elements. Searching here refers to finding an item in the array that meets some specified criterion.

The search is successful if the required element is found. Otherwise, the search is unsuccessful.

Based on the type of search operation, these algorithms are generally classified into two categories:

1. Sequential Search(Linear Search)

In this, the list or array is traversed sequentially and every element is checked.

For example: Linear Search.

Search = 33

Linear Search complexity

In general time complexity of linear search is O(n) because every element in the array is compared only once.

Case	Time Complexity	Space complexity
Best Case	O(1)	O(1)
Average Case	O(n)	O(1)
Worst Case	O(n)	O(1)

2. Binary Search (Interval Search)

- These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half.
- Linear search is easy but the time complexity is O(n) which is reduced by binary search to O(Logn).
- When the data is sorted,(i.e. in numerical or alphabetical order), you can use a much more efficient algorithm called a Binary Search.
- This search algorithm works on the principle of divide and conquers.
- Binary Search Algorithm can be implemented in the following two ways:
  1. **Iterative Method**
  2. **Recursive Method**

Binary Search complexity

Time complexity and space complexity of linear search in the best case, average case, and worst case is given in the following table. In general time complexity of linear search is O(Logn).

Case	Time Complexity	Space complexity
Best Case	**O(1)**	**O(1)**
Average Case	**O(Logn)**	**O(1)**
Worst Case	**O(Logn)**	**O(1)**

Sorting

- Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.
- Sorting algorithms are often classified by :
  a) **Computational complexity**
  b) **Memory Utilization**
  c) **Stability - Maintaining relative order of records of equal keys.**
  d) **No. of comparisons.**
  e) **Methods applied like Insertion, exchange, selection, merging etc.**

- **Sorting is a process of linear ordering of list of objects.**
  Example:- Consider an array;
  int A[10] = { 9, 6, 10, 5, 30, 35, 34, 12, 28, 19)
  The Array sorted in ascending order will be given as;
  A[10] = { 5, 6, 9, 10, 12, 19, 28, 30, 34, 35 }

Sorting Algorithm

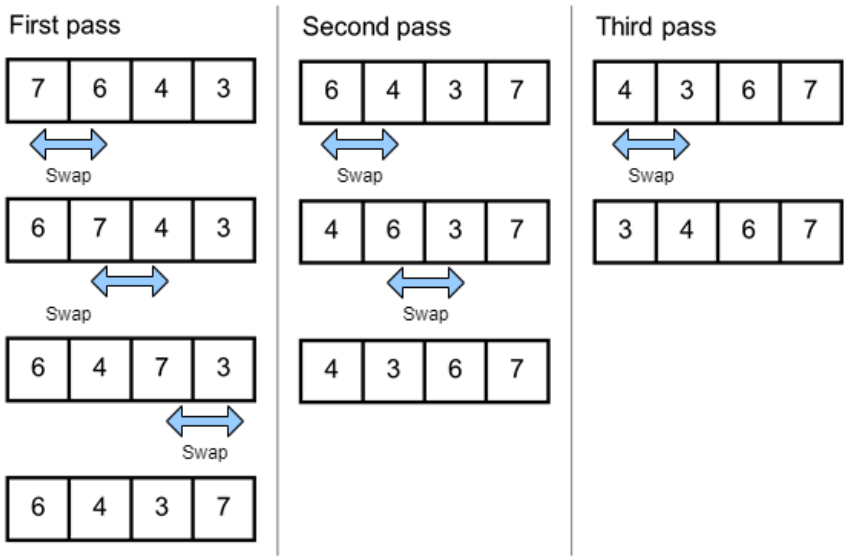
- **A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.**
- **There are some common sorting techniques are:**
  1. Bubble Sort
  2. Insertion Sort
  3. Selection sort
  4. Merge sort
  5. Quick sort
  6. Heap Sort
  7. Shell Sort
  8. Counting sort
  9. Radix sort
  10. Bucket sort

Bubble Sort

- Bubble Sort is the sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
- Bubble sort is stable, in-place, adoptive, internal and comparison based sorting algorithm.
- Bubble sort is easy sorting algorithm however it has a high time complexity $O(n^2)$.
- Bubble sort is an easy algorithm based on going through the array and swapping pairs of numbers. The algorithm works as follows:

1. Starting with the first element, compare it to the second element. If they are out of order, swap them.
2. Compare the 2nd and 3rd element in the same way. Then the 3rd and 4th, etc. (So, go through the array once doing pair-wise swaps.)
3. Repeat this N time, where N is the number of elements in the array.

Example:- consider an array A[4]= {7,6,4,3} and sort this by using bubble sort.



Complexity and stability of Bubble sort

Case	Time Complexity	Space complexity
Best Case	O(n)	O(1)
Average Case	$O(n^2)$	O(1)
Worst Case	$O(n^2)$	O(1)

Insertion Sort

- Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and placed at the correct position in the sorted part.
- Insertion sort is fast and best suitable either when the problem size is small (because it has low overhead) or when the data is nearly sorted (because it is adaptive).
- Insertion sort is stable, in-place, adoptive, internal and comparison based sorting algorithm.
- The overall recurrence relation of recursive insertion sort is given by T(n) = T(n-1) + n.

Insertion sort stability and complexity

The time complexity of insertion sort in best case, average case, and worst case. We will also see the space complexity of insertion sort.

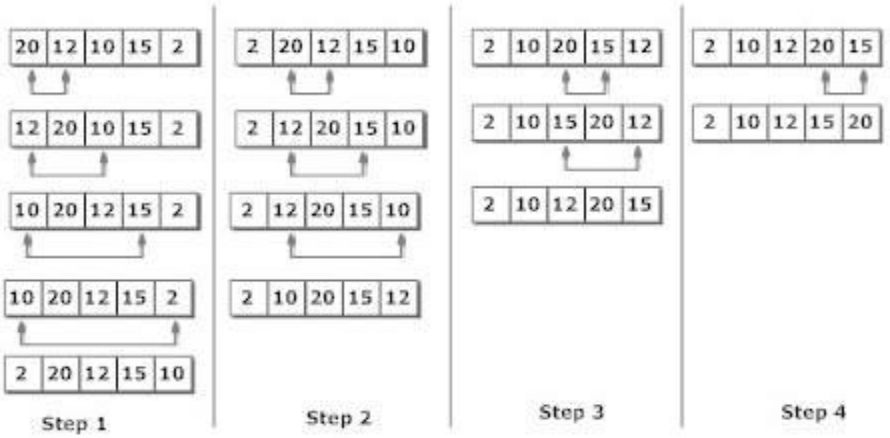
Case	Time Complexity	Space complexity
Best Case	O(n)	O(1)
Average Case	$O(n^2)$	
Worst Case	$O(n^2)$	

Selection Sort

- The selection sort improves on the bubble sort by making only one exchange for every pass through the list.
- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array:
  - The subarray which is already sorted
  - Remaining subarray which is unsorted
- In every iteration/pass of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.
- The selection sort has the property of minimizing the number of swaps. Therefore, it is the best choice when the cost of swapping is high.
- Selection sort is non- stable, in-place, non-adoptive and comparison based sorting algorithm.

Example:- To understand the working of the Selection sort algorithm, let's take an unsorted array.

A[5]= {20,12,10,15,2}



Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

Case	Time Complexity	Space complexity
Best Case	$O(n^2)$	O(1)
Average Case	$O(n^2)$	O(1)
Worst Case	$O(n^2)$	O(1)

Merge sort

- Merge sort uses the "divide and conquer" strategy which divides the array or list into numerous sub arrays and sorts them individually and then merges into a complete sorted array.
- Merge sort performs faster than other sorting methods and also works efficiently for smaller and larger arrays likewise.
- Merge sort is the sorting technique that follows the divide and conquer approach.
- Merge sort is stable, out-place, non-adoptive and external sorting algorithm.
- The recursive formula of merge sort is T(n) = 2T(n/2) + Θ(n).

Merge sort complexity

Case	Time Complexity	Space Complexity
Best Case	O(n*logn)	O(n)
Average Case	O(n*logn)	
Worst Case	O(n*logn)	

Quick sort

- Quicksort is a sorting algorithm that is often faster than most of the types of sorts, including merge sort However, though it is often fast, it is not always fast, and can
- degrade significantly given the wrong conditions.
- The recursive formula of quick sort is T(n) = T(n-1) + Θ(n)
- Quick sort is another divide and conquer algorithm.
- The basic idea: Divide a list into two smaller sublists: the low elements and the high elements. Then, recursively sort the sublists.
- **Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.
- **Conquer:** Recursively, sort two subarrays with Quicksort.
- **Combine:** Combine the already sorted array.

Choosing the pivot:

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows:-

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Quick sort complexity

Case	Time Complexity	Space Complexity
Best Case	O(n*logn)	O(nlogn)
Average Case	O(n*logn)	
Worst Case	$O(n^2)$	

Heap Sort

- A heap data structure where the tree is a complete binary tree is referred to as a binary heap.
- Heaps are mainly of two types:-max heap and min heap.
- In a max heap, the value of a node is always >= the value of each of its children.
- In a min heap, the value of a parent is always <= the value of each of its children.
- Root element: In a max heap, the element at the root will always be the maximum. In a min heap, the root element will always be the smallest. The heap sort algorithm takes advantage of this property to sort an array using heaps.
- Heap sort is non-stable, in-place, adoptive, internal and comparison based sorting algorithm.
- Heap sort is an efficient comparison-based sorting algorithm that:
  - Creates a heap from the input array.
  - Then sorts the array by taking advantage of a heap's properties.
- **After the heap formation using the heapify method, the sorting is done by:**
  1. Swapping the root element with the last element of the array and decreasing the length of the heap array by one. (In heap representation, it is equivalent to swapping the root with the bottom-most and right-most leaf and then deleting the leaf.)
  2. Restoring heap properties (reheapification) after each deletion, where we need to apply the heapify method only on the root node. The subtree heaps will still have their heap properties intact at the beginning of the process.
  3. Repeating this process until every element in the array is sorted: Root removal, its storage in the position of the highest index value used by the heap, and heap length decrement.
- On a max heap, this process will sort the array in ascending order.
- On a min heap, this process will sort in descending order.

Complexity Heap Sort

Total Time Complexity of Heap Sort is O(Nlog(N))
Time for creating a MaxHeap + Time for getting a sorted array out of a MaxHeap =O(N) + O(Nlog(N)) =O(Nlog(N))

Case	Time Complexity	Space Complexity
Best Case	O(n*logn)	O(1)
Average Case	O(n*logn)	
Worst Case	O(nlogn)	

List of complexity of all the sorting algorithm

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Bucket Sort	Ω(n +k)	θ(n +k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n + k)
Count Sort	Ω(n +k)	θ(n +k)	O(n +k)	O(k)
Shell Sort	Ω(n)	θ(n log(n))	O(n log(n))	O(1)