

Memory Allocation	4
What is Data Structure:.....	4
Types of Data Structures	5
Primitive Data structure	5
Non-Primitive Data structure	5
Linear Data Structure.....	5
Non-Linear Data Structure	5
Data structures can also be classified as:	5
Major Operations	5
Abstract data types.	6
Advantages of Data structures.....	6
Data	6
Record	6
File	6
Attribute and Entity	6
Algorithm.....	7
Characteristics of an Algorithm	7
Why do we need Algorithms?.....	7
Example of Algorithm	7
Approach & type of Algorithm.....	8
Brute force algorithm:	8
Recursive Algorithm	8
Backtracking Algorithm	8
Searching Algorithm	8
Sorting Algorithm.....	8
Hashing Algorithm	8
Divide and Conquer Algorithm.....	8
Greedy Algorithm	8
Dynamic Programming Algorithm:	9
Randomized Algorithm	9
Categories of Algorithm.....	9
Sequence:.....	9
Selection:	10
Iteration:	10
Categories of algorithms	10

Algorithm Complexity - Time And Space Complexity:.....	12
Time Complexity:.....	12
Space Complexity:	15
Asymptotic Notations of algorithm	17
Big-O Notation (O-notation) worst case:	18
Omega Notation (Ω -notation) best case.....	18
Theta Notation (Θ -notation) - average case	19
Stable and Unstable Algorithm	20
in-place (internal) and not in-place(external) algorithm	20
Type of Algorithm.....	20
Search Algorithm	21
Linear Search	21
How Linear Search Algorithm Work?	21
Complexity Analysis of Linear Search:	23
Advantages of Linear Search:	23
Drawbacks of Linear Search:	23
When to use Linear Search?	23
Algorithm of Linear search	23
Binary Search.....	24
Complexity Analysis of Binary Search:	26
Advantages of Binary Search:.....	26
Drawbacks of Binary Search:	26
Applications of Binary Search:.....	27
Algorithm of binary search	28
Sort Algorithm	29
Bubble sort.....	29
Insertion sort	31
Selection sort	34
Quick Sort	36
Merge Sort	42
Pointer.....	46
Structure	47
Array in Data Structure	48
2D Array	49
Linked List.....	49
Stack.....	53

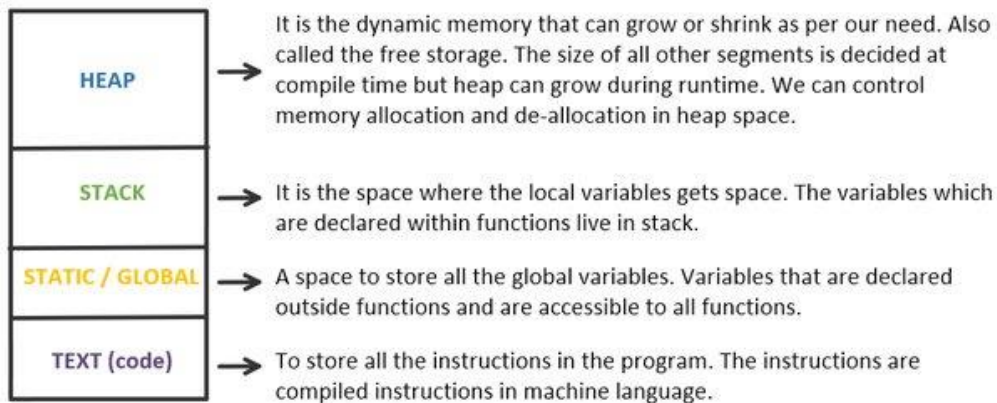
Queue	56
Tree.....	60
Types of Tree data structures:.....	62
Basic Operation Of Tree Data Structure:.....	62
Implementation of Tree	63
BINARY TREE:	63
Tree Traversal Techniques	64
Inorder Traversal - Left->Root->Right (2 time count)	64
Preorder Traversal - Root->Left->Right (1 time count)	65
Postorder Traversal- Left → Right → Root	65
Breadth First Traversal	66
Binary Search Tree(BST):-.....	67
Full/ proper/ strict Binary tree	67
Complete Binary Tree	68
Perfect Binary Tree	68
Degenerate Binary Tree.....	68
AVL TREEE (BALANCED BST)	69
Graph:.....	72

DATA STRUCTURE AND ALGORITHM

Memory Allocation

4 types of memory allocation .

1. STACK – Local variable ,Function
2. HEAP -Pointer ,object
3. Static/Global – global function (you can declare a function global by typing static in front of data type like (static int y=10)
4. CODE /INSTRUCTION



Tridib Samanta

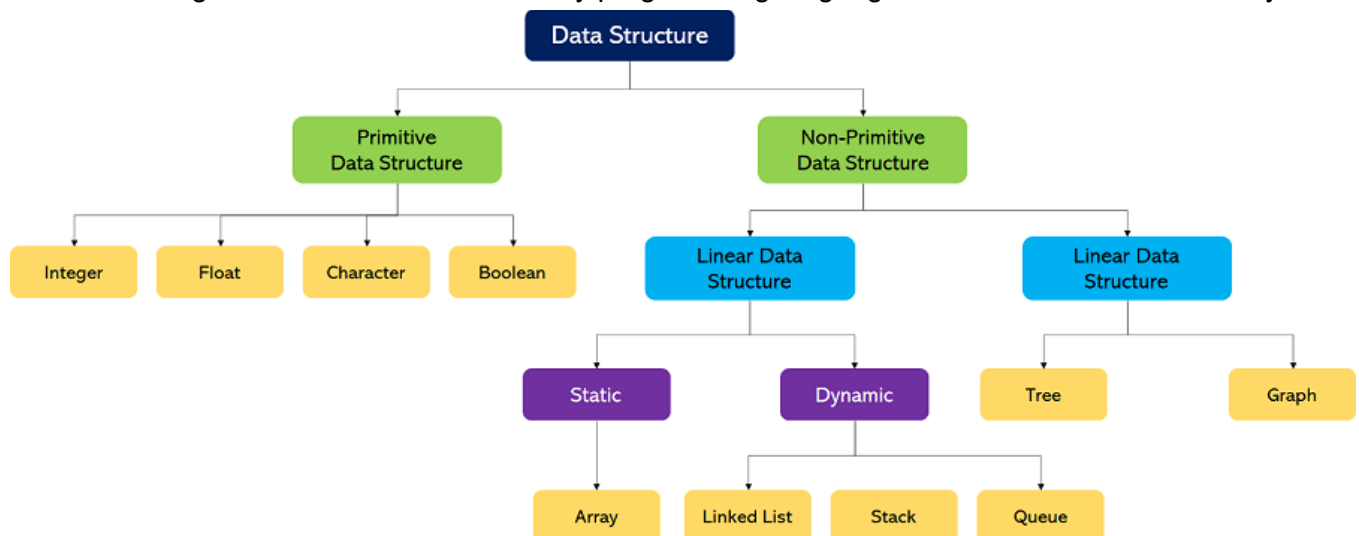
What is Data Structure:

A data structure is a storage that is used to store and organize data.

It is a way of arranging data so that it can be accessed and updated efficiently.

Organized data in a memory .

It is a set of algorithm that we can use in any programming languages to structure data in memory.



Types of Data Structures

There are two types of data structures:

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

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

1. Linear data structure
2. Non-linear data structure

Linear Data Structure

The arrangement of data in a sequential manner is known as a linear data structure.

The data structures used for this purpose are **Arrays, Linked list, Stacks, and Queues**.

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

Non-Linear Data Structure

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

Data structures can also be classified as:

Static data structure: It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.

Dynamic data structure: It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

Searching: We can search for any element in a data structure.

Sorting: We can sort the elements of a data structure either in an ascending or descending order.

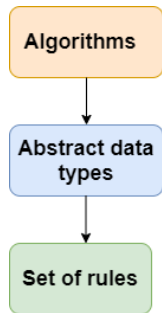
Insertion: We can also insert the new element in a data structure.

Updation: We can also update the element, i.e., we can replace the element with another element.

Deletion: We can also perform the delete operation to remove the element from the data structure.

Abstract data types.

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



An ADT tells **what** is to be done and data structure tells **how** it is to be done.

In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part.

Advantages of Data structures

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

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

Abstraction: The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Data

Data can be defined as the elementary value and collection of value

Ex. Students name and its ID are data about student

Record

Record can be defined as collection of various data item

Ex .student entity name address course and marks can ne grouped together to form record

File

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

Ex. if there are 60 employee in class then there will be 20 record in related file where record contains info of employee

Attribute and Entity

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

Algorithm

An algorithm is a Step by Step process to solve a problem, where each step indicates an intermediate task. Algorithm contains finite number of steps that leads to the solution of the problem.

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

Characteristics of an Algorithm

1. **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
2. **Output:** We will get 1 or more output at the end of an algorithm.
3. **Unambiguity:** the instructions in an algorithm should be clear and simple.
4. **Finiteness:** the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
5. **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
6. **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Why do we need Algorithms?

It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.

Example of Algorithm

We will write an algorithm to add two numbers entered by the user.

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., $\text{sum} = a + b$.

Step 5: Print sum

Step 6: Stop

Approach & type of Algorithm

The following are the approaches used after considering the theoretical and practical importance of designing an algorithm:

Brute force algorithm:

The general logic structure is applied to design an algorithm. It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution.

Such algorithms are of two types

Optimizing:	Sacrificing:
Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.	As soon as the best solution is found, then it will stop.

Recursive Algorithm

A recursive algorithm is based on [recursion](#). In this case, a problem is broken into several sub-parts and called the same function again and again.

Backtracking Algorithm

The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

Searching Algorithm

Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

Sorting Algorithm

Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

Hashing Algorithm

Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

Divide and Conquer Algorithm

This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps:

Divide

Solve

Combine

Greedy Algorithm

In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part.

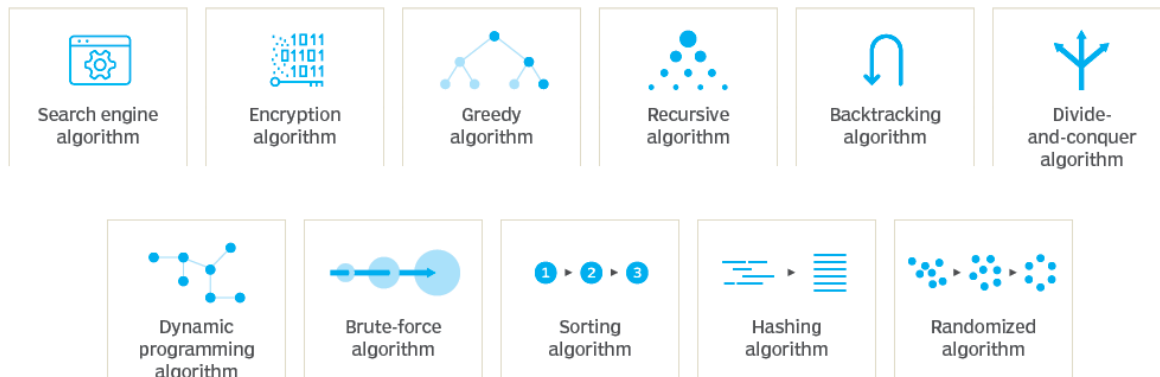
Dynamic Programming Algorithm:

This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

Randomized Algorithm

In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

Types of algorithms



Categories of Algorithm

Based on the different types of steps in an Algorithm, it can be divided into three categories,

1. Sequence
2. Selection
3. Iteration

Sequence:

The steps described in an algorithm are performed successively one by one without skipping any step. The sequence of steps defined in an algorithm should be simple and easy to understand. Each instruction of such an algorithm is executed, because no selection procedure or conditional branching exists in a sequence algorithm.

Example: // adding two numbers

Step 1: start

Step 2: read a,b

Step 3: Sum=a+b

Step 4: write Sum

Step 5: stop

Selection:

In order to solve the problem which involve decision making or option selection, we go for Selection type of algorithm.

```
if(condition) Statement-1;  
else Statement-2;
```

Example1:

```
// Person eligibility for vote  
Step 1 : start  
Step 2 : read age  
Step 3 : if age > = 18 then step_4 else step_5  
Step 4 : write "person is eligible for vote"  
Step 5 : write " person is not eligible for vote"  
Step 6 : stop
```

Example2:

```
// biggest among two numbers  
Step 1 : start  
Step 2 : read a,b  
Step 3 : if a > b then  
Step 4 : write "a is greater than b"  
Step 5 : else  
Step 6 : write "b is greater than a"  
Step 7 : stop
```

Iteration:

Iteration type algorithms are used in solving the problems which involves repetition of statement. In this type of algorithms, a particular number of statements are repeated 'n' no. of times.

Example1:

```
Step 1 : start  
Step 2 : read n  
Step 3 : repeat step 4 until n>0  
Step 4 : (a) r=n mod 10  
          (b) s=s+r  
          (c) n=n/10  
Step 5 : write s  
Step 6 : stop
```

Categories of algorithms

1. **Sort:** Algorithm developed for sorting the items in a certain order.
2. **Search:** Algorithm developed for searching the items inside a data structure.
3. **Delete:** Algorithm developed for deleting the existing element from the data structure.
4. **Insert:** Algorithm developed for inserting an item inside a data structure.
5. **Update:** Algorithm developed for updating the existing element inside a data structure.

1. Write an algorithm for roots of a Quadratic Equation?

Step 1 : start
Step 2 : read a,b,c
Step 3 : if (a= 0) then step 4 else step 5
Step 4 : Write " Given equation is a linear equation "
Step 5 : $d=(b * b) - (4 * a * c)$
Step 6 : if (d>0) then step 7 else step8
Step 7 : Write " Roots are real and Distinct"
Step 8: if(d=0) then step 9 else step 10
Step 9: Write "Roots are real and equal"
Step 10: Write " Roots are Imaginary"
Step 11: stop

2. Write an algorithm to find the largest among three different numbers entered by user

Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b
 If a>c
 Display a is the largest number.
 Else
 Display c is the largest number.
 Else
 If b>c
 Display b is the largest number.
 Else
 Display c is the greatest number.
Step 5: Stop

3. Write an algorithm to find the factorial of a number entered by user.

Step 1: Start
Step 2: Declare variables n, factorial and i.
Step 3: Initialize variables
 factorial ← 1
 i ← 1
Step 4: Read value of n
Step 5: Repeat the steps until i=n
 5.1: factorial ← factorial*i
 5.2: i ← i+1
Step 6: Display factorial
Step 7: Stop

4. Write an algorithm to find the Simple Interest for given Time and Rate of Interest .

Step 1: Start

Step 2: Read P,R,S,T.

Step 3: Calculate $S = (PTR)/100$

Step 4: Print S

Step 5: Stop

Algorithm Complexity - Time And Space Complexity:

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

1. Time Complexity
2. Space Complexity.

Time Complexity:

The time complexity of an algorithm is the amount of time required to complete the execution.

It is denoted by the big O notation.

big O notation is the asymptotic notation to represent the time complexity.

The time complexity is mainly calculated by counting the number of steps to finish the execution.

Instead of measuring actual time required in executing each statement in the code, Time Complexity considers how many times each statement executes.

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

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

Finding Time complexity of any algorithm

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

First step to drop the lower order term

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

$$T(n) = 2n^2$$

Second step drop constant

$$T(n) = 2n^2$$

$$T(n) = n^2$$

So we get the time complexity = $O(n^2)$

Calculate the time complexity of single loop

```
for (int i = 1; i <= n; i++) {  
    x=y+z;  
}
```

our for loop which runs from 1 to N .

we have this expression inside the for loop $X = Y + Z$ which takes constant time to execute now the time complexity will be n times time taken for this expression ($X = Y + Z$) which is a constant let's say C so the time complexity will come out to be C times n and

Cn

in Big O terms we can neglect this constant and simply write the time complexity as Big O(n)

Calculate the time complexity of nested loop

```
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {  
        x=y+z;           //constant time=C  
    }  
}
```

the time complexity so this is expression which takes constant time to execute

now we simply multiply the time complexity of each loop so this is the outer loop which will run n times and

for each run of the outer loop this inner loop will also run n times so the

total running time will be some constant time multiplied by $n * n$ which is equal to n^2 so we can say that the

time complexity is Big O (n^2)

Sequential Statement time complexity

```
1  a=a+b                               // C1  
2  for (int i = 1; i <= n; i++)        // C2 + n  
    { x=y+z; }  
  
3.  for (int j = 1; j <= n; j++)        // C3 + n  
    { d=e+f; }
```

$C1 + C2 + N \quad C3 + N$

~~$C1 + C2 + N \quad C3 + N$~~

~~$-2N$~~

$O(n)$

we have an expression which takes constant time to run let me denote it by $c1$ after that we have this for loop which we have seen before and it will take $c2$ times n time to run then we have another for loop which takes $c3$ into and time to run so when we have sequential statements like these we simply add the time taken for each statement so let me do that in Big O terms we can ignore these constants $c1$ $c2$ and $c3$ and we have the time complexity as Big O of n next let us look at how do we find out time complexity when we have conditional .

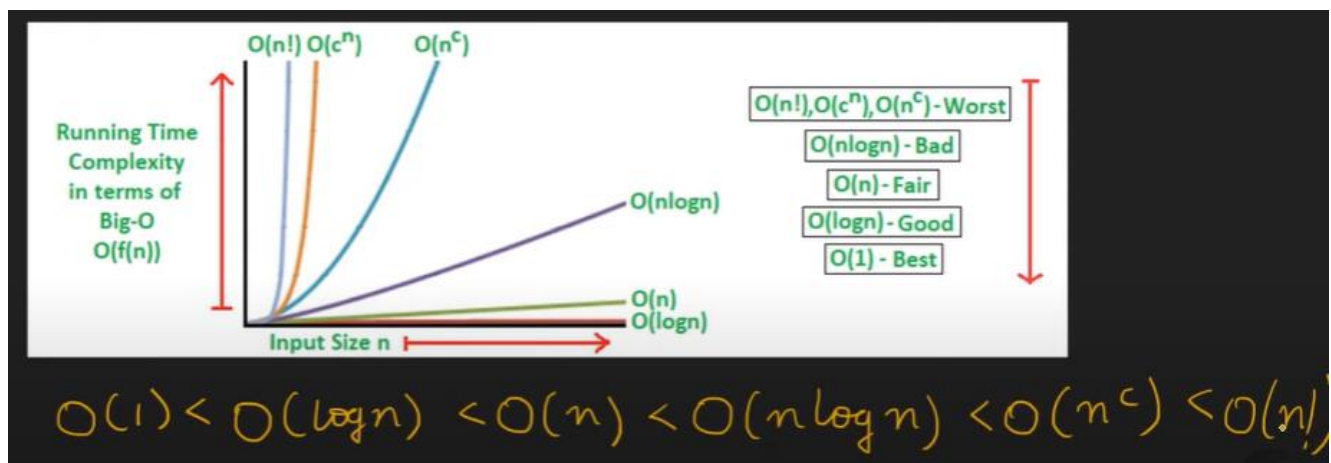
if else statement time complexity

```

if ()
{
    // single loop in if    O( n)
}
Else ()
{
    //nested loop in else O( n²)
}

```

when values in if-else conditions cause maximum number of statements to be executed so let's say that inside this if condition we have a for loop whose time complexity is Big O(n) and inside this else condition we have a nested for loop whose time complexity is Big O(n²) so we consider the else condition as it has a greater time complexity and hence the overall time complexity will come out.



the comparison of time complexities so we have Big O(1) which is the smallest and the best then we have Big O (log n) then comes Big O (n) then it is Big O(n ogn) then we have exponential time complexities and finally we have Big O(n!) factorial which is the worst this is a chart of time complexity of various

This is a chart of time complexity of various sorting algorithms it is highly

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$

Space Complexity:

An algorithm's space complexity is the amount of space required to solve a problem and produce an output.

It is also denoted by big O .

For an algorithm, the space is required for the following purposes:

space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of input

amount of space

```
int a ,b ,c ;
```

```
int z =a+b+c;
```

```
return z ;
```

assuming that an integer takes 4 bytes of storage in memory and in this piece of code we have four integers a b c and z so the total space will come out to be $4 * 4 + 4$ this additional 4 is for the return statement hence this comes out to be 20 bytes so this is a constant space running algorithm there 20bytes

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space:

The extra space required by the algorithm, excluding the input size, is known as an auxiliary space.

The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

Space complexity = Auxiliary space + Input size

Space measurement

two types

1.input space 2.auxiliary space

Input space – input kitna space lega memory main is called input space.

Auxiliary space – program run hote samay extra space le usko auxiliary space bolte hai .

```
int sum(int n)           //input space is constant called order of 1 O(1)
{   int i, sum=0;         //int i, sum=0; is also constant
    for ( i = n; i >=1; i--)
        {   sum=sum+i; }
    return sum; }
```

```
int sum(int n){
    if (n<=0)
    { return 0 ; }
    else{return n+ sum(n-1)}}

```

explain:

```
return n+ sum(n-1)
return 4+ sum(3)
return 3+ sum(2)
return 2+ sum(1)
return 1+ sum(0)
return 0

```

memory allocation hoga stack main
 Auxillary space lgrra hai 5 , input 4 hai

```
1.sum(0) ->0
2.sum(1)-> 1 + sum (0)
3.sum(2)-> 2 + sum (1)
4.sum(3)-> 3 + sum (2)
5.sum(4) -> 4 + sum (3)

```

function **return** karta hai apne parent ko
 sum(0) ko call kia tha sum(1) ne
 toh **return** karega jaha vo call hua tha

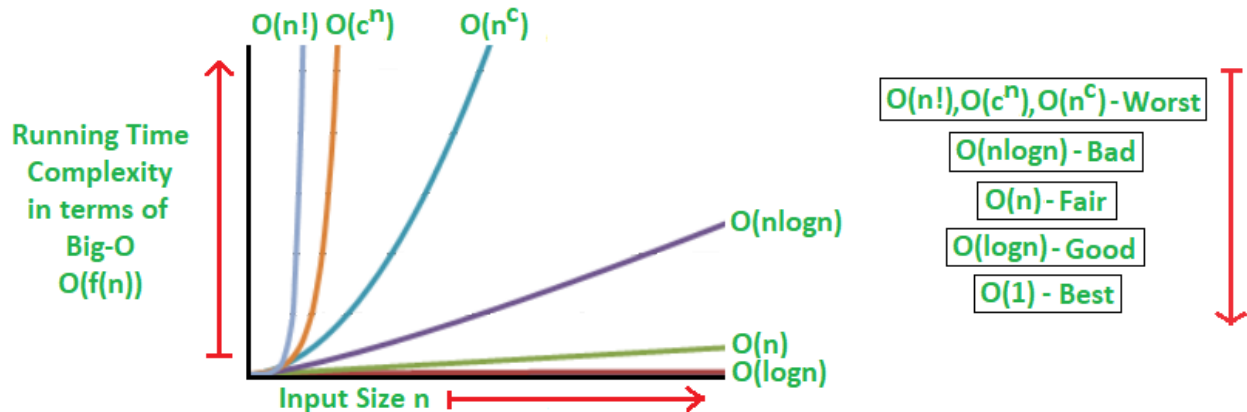
```
sum(0) ->0
sum(1)-> 1 + sum (0) //sum(0) replace hokar ho jaega 0 , than 1 + 0 =1 , sum(1) ki
return value hogai 1
sum(2)-> 2 + sum (1) //sum(1) replace hokar ho jaega 1 , than 2 + 1 =3 ,sum(1) ki return
value hogai 3
sum(3)-> 3 + sum (2)//sum(2) replace hokar ho jaega 2 , than 3 + 3 =6 ,sum(1) ki return
value hogai 6
sum(4) -> 4 + sum (3)//sum(3) replace hokar ho jaega 6 , than 4 + 6 =10 ,sum(1) ki
return value hogai 10

```

input n = 4 so input to constant hi hai . 0(1)
 Auxillary space lgrra hai 5 , input 4 hai 0(n)
space complicity = 0(1) + 0(n) = 0(n)

Asymptotic Notations of algorithm

- 1 Asymptotic Notations are the abstract notion for describing the behavior of algorithm and determine the rate of growth of function .
- 2 Measure of the efficiency of algorithms that don't depend on machine-specific constants
- 3 Don't require algorithms to be implemented
- 4 Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.



Example to build a house

Builder tell you the maximum cost and minimum and average cost

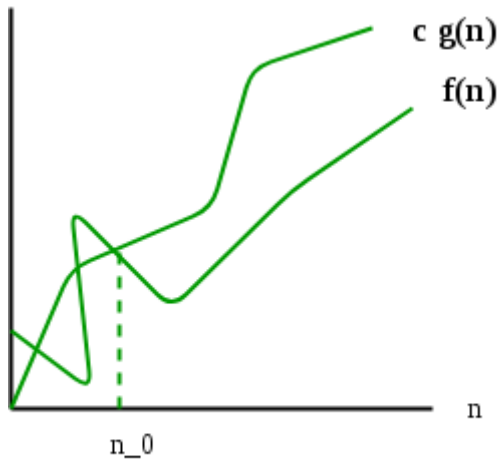
- 1 Maximum cost 15000 $O(n^2)$ upper bound
- 2 Exact cost 10000 $\Theta(n)$
- 3 Minimum cost 8000 $\Omega(\log n)$ lower bound

There are mainly three asymptotic notations:

- 1 Big-O Notation (O -notation) - WORST CASE
- 2 Omega Notation (Ω -notation) - best case
- 3 Theta Notation (Θ -notation) - average case

Big-O Notation (O-notation) worst case:

- 1 Big-O notation represents the upper bound of the running time of an algorithm.
- 2 Therefore, it gives the worst-case complexity of an algorithm.
- 3 It is the most widely used notation for Asymptotic analysis
- 4 It specifies the upper bound of a function.
- 5 The maximum time required by an algorithm or the worst-case time complexity.
- 6 It returns the highest possible output value(big-O) for a given input.
- 7 Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.



$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$

$$f(n) \leq g(n)$$

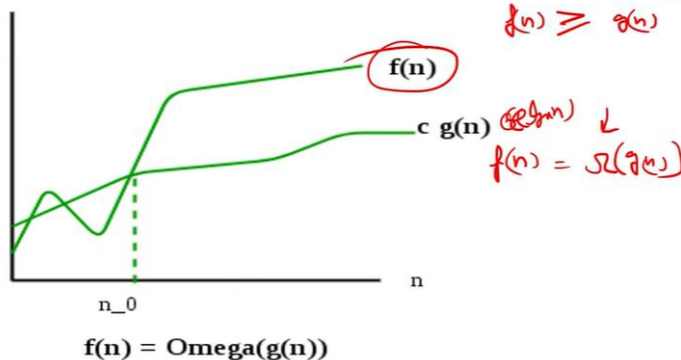
$f(n) \leq O(g(n))$ // jo bada a usko hi sign dena hai

c = constant se fark nahi pdata

Omega Notation (Ω -notation) best case

Ω Notation

- Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
- Ω Notation can be useful when we have lower bound on time complexity of an algorithm.
- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions.
- $\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0 \}$.



$$f(n) \geq c \cdot g(n)$$

$$f(n) \geq \Omega(g(n))$$

// jo chota h usko hi sign dena hai

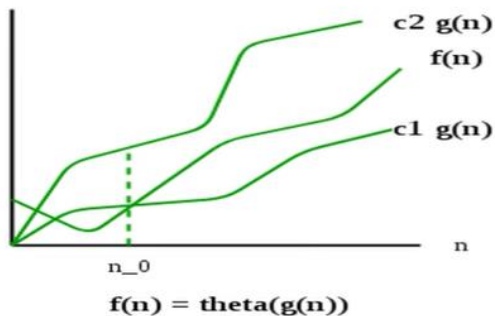
c = constant se fark nahi pdata



Theta Notation (Θ -notation) - average case

Theta Notation

- **Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behaviour.
- For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.
- $\Theta(g(n)) = \{f(n): \text{there exist positive constants } C_1, C_2 \text{ and } n_0 \text{ such that } 0 \leq C_1 * g(n) \leq f(n) \leq C_2 * g(n) \text{ for all } n \geq n_0\}$
- The above definition means, if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $C_1 * g(n)$ and $C_2 * g(n)$ for large values of n ($n \geq n_0$).
- The definition of theta also requires that $f(n)$ must be non-negative for values of n greater than n_0 .



$$C_1 g(n) < f(n) < C_2 g(n)$$

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

Analogy of asymptomatic notation with real numbers

$$f(n) \text{ is } O(g(n)) \quad \rightarrow \quad a \leq b$$

$$f(n) \text{ is } \Omega(g(n)) \quad \rightarrow \quad a \geq b$$

$$f(n) \text{ is } \Theta(g(n)) \quad \rightarrow \quad a = b$$

$$f(n) \text{ is } o(g(n)) \quad \rightarrow \quad a < b$$

$$f(n) \text{ is } \omega(g(n)) \quad \rightarrow \quad a > b$$

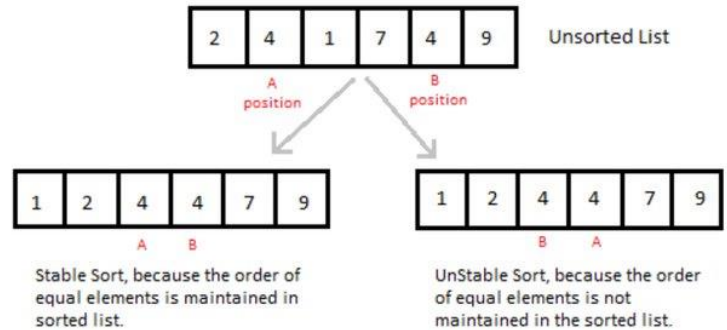
a b

Conclusion

- Most useful notation is Theta, followed by Big O
- the Omega notation is the least used notation among all three

Stable and Unstable Algorithm

A sorting algorithm is said to be stable if two objects with equal key appear in the same order in the sorted output as they appear in the unsorted input .



in-place (internal) and not in-place(external) algorithm

An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'.

However, a small constant extra space used for variables is allowed.

When we don't take extra array to solve the another array based problem .

Type of Algorithm

1. Searching Algorithm
2. Sorting Algorithm

Searching Algorithm

1. Linear Search
2. Binary Search

Sorting Algorithm

<ul style="list-style-type: none">• Selection Sort• Bubble Sort• Insertion Sort• Merge Sort• Quick Sort• Heap Sort• Counting Sort• Radix Sort• Bucket Sort• Bingo Sort Algorithm• ShellSort• TimSort	<ul style="list-style-type: none">• Comb Sort• Pigeonhole Sort• Cycle Sort• Cocktail Sort• Strand Sort• Bitonic Sort• Pancake sorting• BogoSort or Permutation Sort• Gnome Sort	<ul style="list-style-type: none">• Sleep Sort – The King of Laziness• Structure Sorting in C++• Stooge Sort• Tag Sort (To get both sorted and original)• Tree Sort• Odd-Even Sort / Brick Sort• 3-way Merge Sort
---	---	---

Search Algorithm

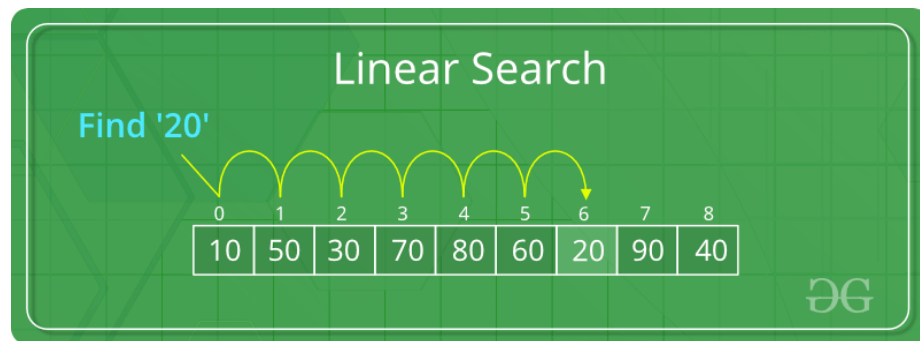
On each day, we search for something in our day to day life.

Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user.

There are mainly two techniques available to search the data in an array:

Linear Search

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.



Linear Search Algorithm

How Linear Search Algorithm Work?

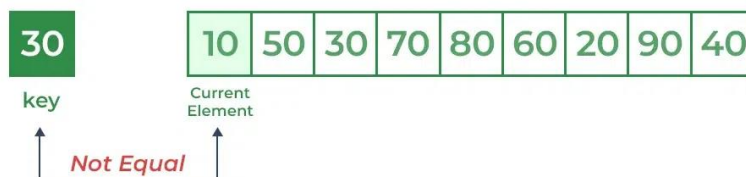
- If any element is found equal to the key, the search is successful and the index of that element is returned.
- If no element is found equal to the key, the search yields “No match found”.

For example:

Consider the array `arr[] = {10, 50, 30, 70, 80, 20, 90, 40}` and `key(we have to find) = 30`

Step 1: Start from the first element (index 0) and compare `key` with each element (`arr[i]`).

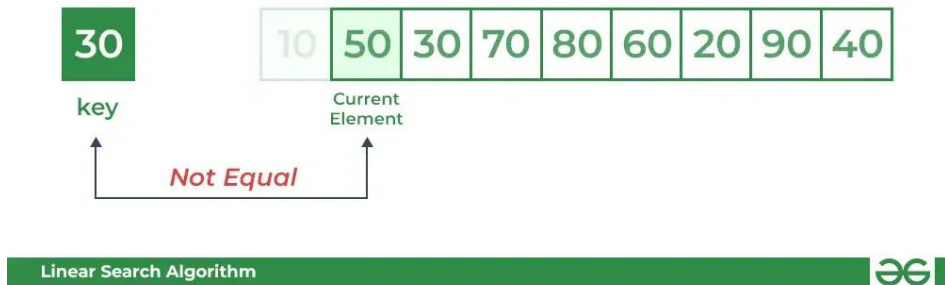
Comparing key with first element `arr[0]`. Since not equal, the iterator moves to the next element as a potential match.



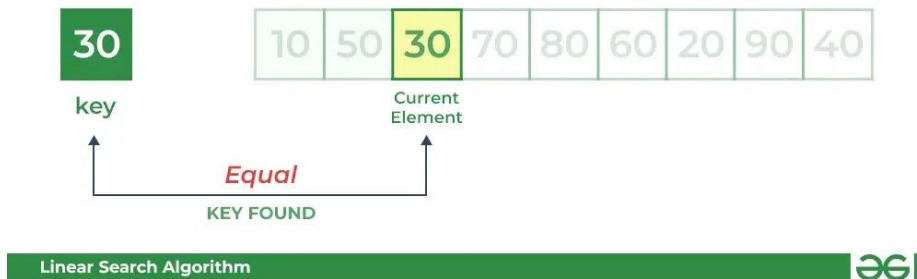
Linear Search Algorithm



Comparing key with next element arr[1]. Since not equal, the iterator moves to the next element as a potential match.



Step 2: Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).



// C++ code to linearly search x in arr[].

```
#include <bits/stdc++.h>
using namespace std;
int search(int arr[], int N, int x)
{ for (int i = 0; i < N; i++)
    if (arr[i] == x)
        return i;
    return -1 }
// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int N = sizeof(arr) / sizeof(arr[0]);
    // Function call
    int result = search(arr, N, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;}
```

Output:Element is present at index 3

Complexity Analysis of Linear Search:

Time Complexity:

Best Case: In the best case, the key might be present at the first index. So the best case complexity is $O(1)$

Worst Case: In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is $O(N)$ where N is the size of the list.

Average Case: $O(N)$

Auxiliary Space: $O(1)$ as except for the variable to iterate through the list, no other variable is used.

Advantages of Linear Search:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

Drawbacks of Linear Search:

- Linear search has a time complexity of $O(N)$, which in turn makes it slow for large datasets.
- Not suitable for large arrays.

When to use Linear Search?

- When we are dealing with a small dataset.
- When you are searching for a dataset stored in contiguous memory.

Algorithm of Linear search

STEP 1 : START
STEP 2 : SET POSITION = 0
STEP 3 : IF POSITION > MAX
STEP 4 : PRINT OVER
STEP 5 : ELSE
STEP 6 : ARRAY[POSITION] == VALUE
STEP 7 : PRINT FOUND
STEP 8 : POSITION + 1
STEP 9 : END

```
Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
Step 1: set pos = -1
Step 2: set i = 1
Step 3: repeat step 4 while i <= n
Step 4: if a[i] == val
    set pos = i
    print pos
    go to step 6
[end of if]
set i = i + 1
[end of loop]
Step 5: if pos = -1
    print "value is not present in the array "
[end of if]
Step 6: exit
```

Binary Search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log N)$.

Conditions for when to apply Binary Search in a Data Structure:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method :The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

$$\text{mid} = (\text{beg} + \text{end})/2$$

$$\text{beg} = 0$$

$$\text{end} = 8$$

$$\text{mid} = (0 + 8)/2 = 4. \text{ So, 4 is the mid of the array.}$$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
 $A[\text{mid}] = 39$
 $A[\text{mid}] < K$ (or, $39 < 56$)
So, $\text{beg} = \text{mid} + 1 = 5$, $\text{end} = 8$
Now, $\text{mid} = (\text{beg} + \text{end})/2 = 13/2 = 6$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
 $A[\text{mid}] = 51$
 $A[\text{mid}] < K$ (or, $51 < 56$)
So, $\text{beg} = \text{mid} + 1 = 7$, $\text{end} = 8$
Now, $\text{mid} = (\text{beg} + \text{end})/2 = 15/2 = 7$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
 $A[\text{mid}] = 56$
 $A[\text{mid}] = K$ (or, $56 = 56$)
So, location = mid
Element found at 7th location of the array

Now, the element to search is found. So algorithm will return the index of the element matched.

Binary search program to using recursion

```
#include <iostream>
using namespace std;
int binarySearch(int a[], int beg, int end, int val)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        /* if the item to be searched is present at middle */
        if(a[mid] == val)
        {
            return mid+1;
        }
        /* if the item to be searched is smaller than middle, then it can only be in
left subarray */
        else if(a[mid] < val)
        {
            return binarySearch(a, mid+1, end, val);
        }
        /* if the item to be searched is greater than middle, then it can only be in
right subarray */
        else
        {
            return binarySearch(a, beg, mid-1, val);
        }
    }
    return -1;
}

int main() {
    int a[] = {10, 12, 24, 29, 39, 40, 51, 56, 70}; // given array
    int val = 51; // value to be searched
    int n = sizeof(a) / sizeof(a[0]); // size of array
    cout<<"size of array : " <<n <<endl;
    int res = binarySearch(a, 0, n-1, val); // Store result
    cout<<"The elements of the array are - ";
    for (int i = 0; i < n; i++)
        cout<<a[i]<<" ";
    cout<<"\nElement to be searched is - "<<val;
    if (res == -1)
        cout<<"\nElement is not present in the array";
    else
        cout<<"\nElement is present at "<<res<<" position of array";
    return 0;
}
```

Here's a step-by-step explanation of the code:

1. The `binarySearch` function takes an array `a`, the beginning index `beg`, the ending index `end`, and the value `val` to be searched.
2. It calculates the `mid` index by finding the average of `beg` and `end`.
3. If the value at `mid` index is equal to `val`, the function returns `mid + 1` (1-indexed position) as the value was found.
4. If the value at `mid` index is less than `val`, the function calls itself recursively with the search range narrowed to the right half of the array.
5. If the value at `mid` index is greater than `val`, the function calls itself recursively with the search range narrowed to the left half of the array.
6. The `main` function initializes the sorted array `a` and the value `val` to be searched.
7. It calculates the size of the array `n` using the `sizeof` operator.
8. It calls the `binarySearch` function to search for the value `val` in the array.
9. The loop in `main` prints the elements of the array.
10. If the `res` variable is -1, it indicates that the value was not found, and the program outputs a message accordingly. Otherwise, it prints the position where the value was found in the array.

Complexity Analysis of Binary Search:

Time Complexity:

- Best Case: $O(1)$
- Average Case: $O(\log N)$
- Worst Case: $O(\log N)$

Auxiliary Space: $O(1)$, If the recursive call stack is considered then the auxiliary space will be $O(\log N)$.

Advantages of Binary Search:

- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Drawbacks of Binary Search:

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

Applications of Binary Search:

- Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
- It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.
- It can be used for searching a database.

// C++ program to implement iterative Binary Search

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
int binarySearch(int arr[], int l, int r, int x)
```

```
{    while (l <= r) {  
        int m = l + (r - l) / 2;  
        if (arr[m] == x)                // Check if x is present at mid  
            return m;  
        // If x greater, ignore left half  
        if (arr[m] < x) {l = m + 1;}  
        // If x is smaller, ignore right half  
        Else { r = m - 1; }    }
```

```
// If we reach here, then element was not present
```

```
    return -1; }
```

```
int main(void)
```

```
{    int arr[] = { 2, 3, 4, 10, 40 };  
    int x = 10;  
    int n = sizeof(arr) / sizeof(arr[0]);  
    int result = binarySearch(arr, 0, n - 1, x);  
    (result == -1)  
        ? cout << "Element is not present in array"  
        : cout << "Element is present at index " << result;  
    return 0; }
```

1. The `binarySearch` function takes an integer array `arr`, the left index `l`, the right index `r`, and the target value `x` to be searched for.

2. The while loop runs as long as the left index l is less than or equal to the right index r .
3. Inside the loop, the middle index m is calculated using integer division of $(l + r) / 2$.
4. If the value at index m (`arr[m]`) is equal to the target value x , then the function returns m , indicating that the element was found at index m .
5. If the value at index m is less than the target value x , it means that the target is in the right half of the current range, so the l index is updated to $m + 1$.
6. If the value at index m is greater than the target value x , it means that the target is in the left half of the current range, so the r index is updated to $m - 1$.
7. If the loop completes and the target value x is not found in the array, the function returns -1 .
8. In the `main` function, an example array `arr` is defined along with the target value x to be searched.
9. The size of the array n is calculated using the `sizeof` operator.
10. The `binarySearch` function is called with the array, left index 0 , right index $n - 1$, and the target value x .
11. Depending on whether the `result` is -1 or not, the program outputs whether the element was found or not, along with its index.

Algorithm of binary search

Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search

Step 1: set `beg` = `lower_bound`, `end` = `upper_bound`, `pos` = -1

Step 2: repeat steps 3 and 4 while `beg` \leq `end`

Step 3: set `mid` = $(beg + end) / 2$

Step 4: if `a[mid]` = `val`

set `pos` = `mid`

print `pos`

go to step 6

else if `a[mid]` $>$ `val`

set `end` = `mid` - 1

else

set `beg` = `mid` + 1

[end of if]

[end of loop]

Step 5: if `pos` = -1

print "value is not present in the array"

[end of if]

Step 6: exit

Sort Algorithm

Bubble sort

Bubble sort works by repeatedly swapping the adjacent elements if they are in the wrong order

$L > R$ = wrong side

If $L > R$ then SWAP



1st iteration - $n-1$
2nd iteration - $n-2$
and so on...

i^{th} iteration - $n-i$

Complexity Analysis of Bubble Sort

Time Complexity: $O(N^2)$

Auxiliary Space: $O(1)$

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort

- Bubble sort has a time complexity of $O(N^2)$ which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

What is the Boundary Case for Bubble sort?

Bubble sort takes minimum time (Order of n) when elements are already sorted. Hence it is best to check if the array is already sorted or not beforehand, to avoid $O(N^2)$ time complexity.

Does sorting happen in place in Bubble sort?

Yes, Bubble sort performs the swapping of adjacent pairs without the use of any major data structure. Hence Bubble sort algorithm is an in-place algorithm.

Is the Bubble sort algorithm stable?

Yes, the bubble sort algorithm is stable.

```
#include<iostream>
using namespace std;
int main()
{
    int n ;
    cin >>n;
    int a[n];
    for (int i = 0; i < n; i++)
    {
        cin>>a[i];
    }

    int count =1;
    while (count <n)
    {
        for (int i = 0; i < count; i++)
        {
            if (a[i] > a[i+1]){
                int temp=a[i];
                a[i]=a[i+1];
                a[i+1]=temp; }
        }
        count++;
    }
    for (int i = 0; i < n; i++)
    {
        cout<<a[i] <<"\t";
    }
    return 0;
}
```

Insertion sort

It is a simple sorting algo which sort array element by shifting one by one

Consider an example: arr[]: {12, 11, 13, 5, 6}

12	11	13	5	6
----	----	----	---	---

First Pass:

Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6
----	----	----	---	---

Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.

So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6
----	----	----	---	---

Second Pass:

Now, move to the next two elements and compare them

11	12	13	5	6
----	----	----	---	---

Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

Now, two elements are present in the sorted sub-array which are 11 and 12

Moving forward to the next two elements which are 13 and 5

11	12	13	5	6
----	----	----	---	---

Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6
----	----	---	----	---

After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6
----	---	----	----	---

Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6
---	----	----	----	---

Here, 5 is at its correct position

Fourth Pass:

Now, the elements which are present in the sorted sub-array are 5, 11 and 12

Moving to the next two elements 13 and 6

5	11	12	13	6
---	----	----	----	---

Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13
---	----	----	---	----

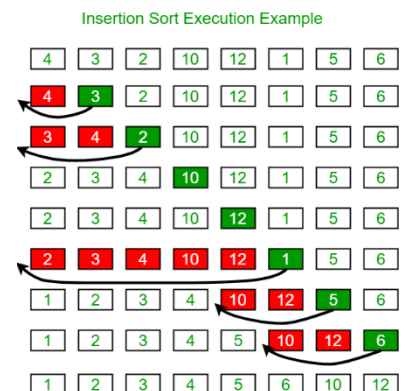
Now, 6 is smaller than 12, hence, swap again

5	11	6	12	13
---	----	---	----	----

Here, also swapping makes 11 and 6 unsorted hence, swap again

5	6	11	12	13
---	---	----	----	----

Finally, the array is completely sorted.



Time Complexity of Insertion Sort

The **worst-case** time complexity of the Insertion sort is $O(N^2)$

The **average case** time complexity of the Insertion sort is $O(N^2)$

The time complexity of the **best case** is $O(N)$.

Space Complexity of Insertion Sort

The auxiliary space complexity of Insertion Sort is $O(1)$

Q1. What are the Boundary Cases of the Insertion Sort algorithm?

Insertion sort takes the maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Q2. What is the Algorithmic Paradigm of the Insertion Sort algorithm?

The Insertion Sort algorithm follows an incremental approach.

Q3. Is Insertion Sort an in-place sorting algorithm?

Yes, insertion sort is an in-place sorting algorithm.

Q4. Is Insertion Sort a stable algorithm?

Yes, insertion sort is a stable sorting algorithm.

Q5. When is the Insertion Sort algorithm used?

Insertion sort is used when number of elements is small. It can also be useful when the input array is almost sorted, and only a few elements are misplaced in a complete big array.

// C++ program for insertion sort

```
#include <bits/stdc++.h>
using namespace std;
```

```
// Function to sort an array using
// insertion sort
```

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of their
        // current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

```
// A utility function to print an array
// of size n
```

```
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

```
// Driver code
```

```
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int N = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, N);
    printArray(arr, N);

    return 0;
}
```

Selection sort

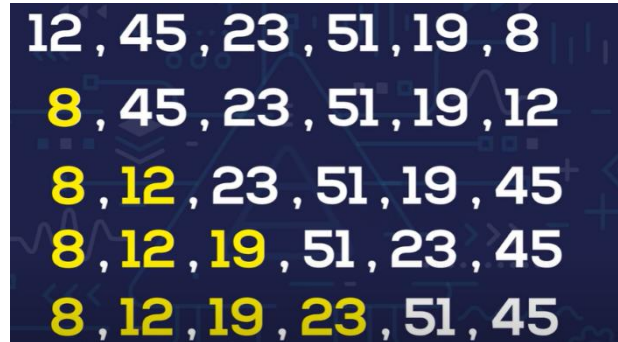
Finding the minimum element in unsorted array and swap it with element in the beginning

```
#include <bits/stdc++.h>
using namespace std;
// Function for Selection sort
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    // One by one move boundary of
    // unsorted subarray
    for (i = 0; i < n - 1; i++) {
        // Find the minimum element in
        // unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        // Swap the found minimum element
        // with the first element
        if (min_idx != i)
            swap(arr[min_idx], arr[i]);
    }
}

// Function to print an array
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++) {
        cout << arr[i] << " ";
        cout << endl;
    }
}

int main()
{
    int arr[] = { 64, 25, 12, 22, 11 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function Call
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```



Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is $O(N^2)$ as there are two nested loops:
One loop to select an element of Array one by one = $O(N)$
Another loop to compare that element with every other Array element = $O(N)$
Therefore overall complexity = $O(N) * O(N) = O(N*N) = O(N^2)$

Auxiliary Space: $O(1)$ as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than $O(N)$ swaps and can be useful when memory writing is costly.

Disadvantages of the Selection Sort Algorithm

Selection sort has a time complexity of $O(n^2)$ in the worst and average case.

Does not work well on large datasets.

Does not preserve the relative order of items with equal keys which means it is not stable.

Q1. Is Selection Sort Algorithm stable?

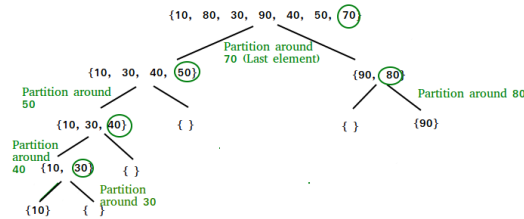
The default implementation of the Selection Sort Algorithm is not stable. However, it can be made stable. Please see the stable Selection Sort for details.

Q2. Is Selection Sort Algorithm in-place?

Yes, Selection Sort Algorithm is an in-place algorithm, as it does not require extra space.

Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.



Choice of Pivot:

1. first element as a pivot.
2. last element as a pivot
3. random element as a pivot.
4. middle as the pivot.

Partition Algorithm:

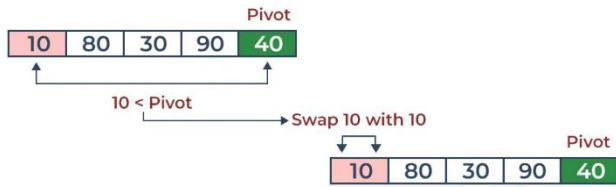
The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as i .

While traversing, if we find a smaller element, we swap the current element with $arr[i]$. Otherwise, we ignore the current element.

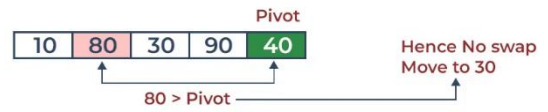
Consider: $arr[] = \{10, 80, 30, 90, 40\}$.

Pivot = 40

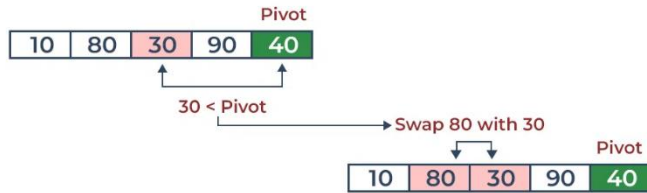
1. $10 < 40$ swap
2. $80 > 40$ no swap
3. $30 < 40$ swap swap with 80
4. $90 > 40$ no swap



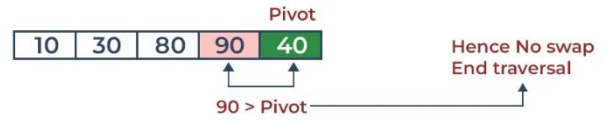
Quick Sort



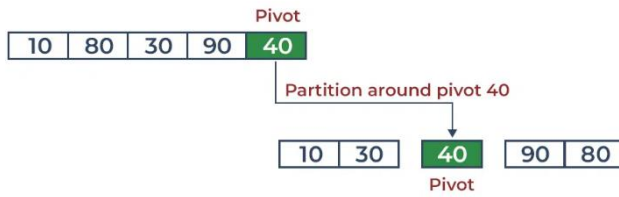
Quick Sort



Quick Sort



Quick Sort



Quick Sort



Quick Sort



```

#include <bits/stdc++.h>
using namespace std;
// This function takes last element as pivot,
// places the pivot element at its correct position
// in sorted array, and places all smaller to left
// of pivot and all greater elements to right of pivot
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];           // Choosing the pivot
    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {

            // Increment index of smaller element
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// The main function that implements QuickSort
// arr[] --> Array to be sorted,
// low --> Starting index,
// high --> Ending index
void quickSort(int arr[], int low, int high)
{
    if (low < high) {

        // pi is partitioning index, arr[p]
        // is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    quickSort(arr, 0, N - 1);
    cout << "Sorted array: " << endl;
    for (int i = 0; i < N; i++)
        cout << arr[i] << " ";
    return 0;
}

```

Complexity Analysis of Quick Sort:

Time Complexity:

- **Best Case:** $\Omega(N \log(N))$
- **Average Case:** $\theta(N \log(N))$
- **Worst Case:** $O(N^2)$

Auxiliary Space: $O(1)$, if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make $O(N)$.

Advantages of Quick Sort:

1. It is a divide-and-conquer algorithm that makes it easier to solve problems.
2. It is efficient on large data sets.
3. It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

1. It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
2. It is not a good choice for small data sets.
3. It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Example of QUICK SORT

6	3	9	5	2	8**
---	---	---	---	---	-----

We choose pivot last element = 8**

Condition if element < pivot = swap

So .

$6 < 8^{**}$ = shift to left

$3 < 8^{**}$ = shift to left

$9 < 8^{**}$ = shift to right

$5 < 8^{**}$ = shift to left

$2 < 8^{**}$ = shift to right

```

#include <iostream>

// Function to partition the array and return the index of the pivot
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose the last element as the pivot
    int i = low - 1; // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1; // Return the index of the pivot
}

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

// Utility function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    std::cout << "Original array: ";
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    std::cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}

```


1. The program includes the necessary header:

- `#include <iostream>`: This provides input-output stream functionality.

2. The `partition` function:

- Takes an array, a low index, and a high index as arguments.
- Chooses the last element (at the `high` index) as the pivot.
- Initializes an index `i` to track smaller elements.
- Loops through elements from `low` to `high - 1`:
 - If an element is smaller than the pivot:
 - Increments `i`.
 - Swaps the smaller element with the element at index `i`.
- Swaps the pivot element to its correct position (after smaller elements).
- Returns the index of the pivot after rearrangement.

3. The `quickSort` function:

- Takes an array, a low index, and a high index as arguments.
- If `low` is less than `high`:
 - Calls the `partition` function to find the index of the pivot.
 - Recursively calls `quickSort` for the subarray on the left of the pivot (from `low` to `pivotIndex - 1`).
 - Recursively calls `quickSort` for the subarray on the right of the pivot (from `pivotIndex + 1` to `high`).

4. The `printArray` function:

- Takes an array and its size as arguments.
- Loops through the array elements and prints them separated by spaces.

5. The `main` function:

- Initializes an example array `arr`.
- Calculates the size of the array using the `sizeof` operator.
- Prints the original array using the `printArray` function.
- Calls `quickSort` to sort the array.
- Prints the sorted array using the `printArray` function.
- Returns `0` to indicate successful program execution.

6. Quick Sort Algorithm:

- Chooses the last element as the pivot.
- Partitions array around the pivot, placing smaller elements to its left and greater elements to its right.
- Recursively sorts the subarrays on both sides of the pivot.
- Eventually, the entire array becomes sorted.

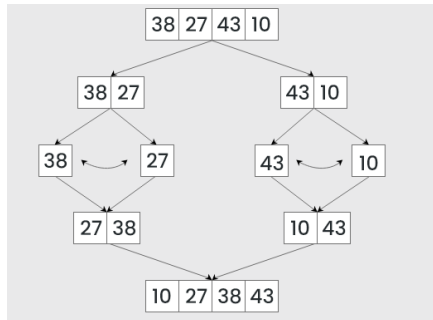
7. Running the Program:

- The original array is printed.
- The array is sorted using the Quick Sort algorithm.
- The sorted array is printed.

Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

Not in-place



How does Merge Sort work?

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

1. MERGE_SORT(arr, beg, end)
2. **if** beg < end
3. set mid = (beg + end)/2
4. MERGE_SORT(arr, beg, mid)
5. MERGE_SORT(arr, mid + 1, end)
6. MERGE (arr, beg, mid, end)
7. end of **if**
8. END MERGE_SORT

The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg...mid]** and **A[mid+1...end]**, to build one sorted array **A[beg...end]**. So, the inputs of the **MERGE** function are **A[]**, **beg**, **mid**, and **end**.

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

    while (j < n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}
```

Working of Merge sort Algorithm

To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.

divide	12	31	25	8	32	17	40	42
--------	----	----	----	---	----	----	----	----

Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

divide	12	31	25	8	32	17	40	42
--------	----	----	----	---	----	----	----	----

Now, again divide these arrays to get the atomic value that cannot be further divided.

divide	12	31	25	8	32	17	40	42
--------	----	----	----	---	----	----	----	----

Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.

merge	12	31	8	25	17	32	40	42
-------	----	----	---	----	----	----	----	----

In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.

merge	8	12	25	31	17	32	40	42
-------	---	----	----	----	----	----	----	----

Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -

8	12	17	25	31	32	40	42
---	----	----	----	----	----	----	----

Now, the array is completely sorted.

Merge sort complexity

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

Time Complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
Worst Case	$O(n \cdot \log n)$

Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **$O(n \cdot \log n)$** .

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **$O(n \cdot \log n)$** .

Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **$O(n \cdot \log n)$** .

Space Complexity

Space Complexity	$O(n)$
Stable	YES

The space complexity of merge sort is $O(n)$. It is because, in merge sort, an extra variable is required for swapping

Applications of Merge Sort:

Sorting large datasets: Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of $O(n \log n)$.

External sorting: Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.

Advantages of Merge Sort:

Stability: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.

Guaranteed worst-case performance: Merge sort has a worst-case time complexity of $O(N \log N)$, which means it performs well even on large datasets.

Parallelizable: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.

Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.

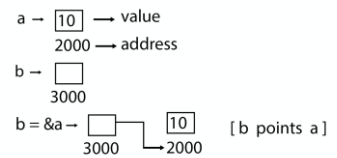
Not always optimal for small datasets: For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

Pointer

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

Pointer improves the performance for repetitive process such as:

- 1 Traversing String
- 2 Lookup Tables
- 3 Control Tables
- 4 Tree Structures



```
#include<iostream>
using namespace std;
int main()
{
    int a = 5;
    int *b;
    b = &a;
    cout<<"value of a " <<a <<endl; //5
    cout<<"value of a " << *(&a)<<endl; //5
    cout<<"value of a " <<*b<<endl; //4
    cout<<"address of a " <<&a<<endl; //0x61ff0c
    cout<<"address of a " <<b<<endl; //0x61ff0c
    cout<<"address of b : " <<&b<<endl; //0x61ff0c
    cout<<"value of b = address of a = " <<b<<endl; //0x61ff0c
    return 0;
}
```

Pointer to Pointer

```
#include <stdio.h>
int main( ) {
    int a = 5;
    int *b;
    int **c;
    b = &a;
    c = &b;
    printf ("value of a = %d\n", a); //5
    printf ("value of a = %d\n", *(&a)); //5
    printf ("value of a = %d\n", *b); //5
    printf ("value of a = %d\n", **c); //5
    printf ("value of b = address of a = %u\n", b); // 2831685116
    printf ("value of c = address of b = %u\n", c); // 2831685120
    printf ("address of a = %u\n", &a); // 2831685116
    printf ("address of a = %u\n", b); // 2831685116
    printf ("address of a = %u\n", *c); // 2831685116
    printf ("address of b = %u\n", &b); //// 2831685120
    printf ("address of b = %u\n", c); //// 2831685120
    printf ("address of c = %u\n", &c); //// 2831685124
    return 0;
}
```

Structure

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

It allows different variables to be accessed by using a single pointer to the structure.

Syntax

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

Advantages

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

```
#include<iostream>
using namespace std;

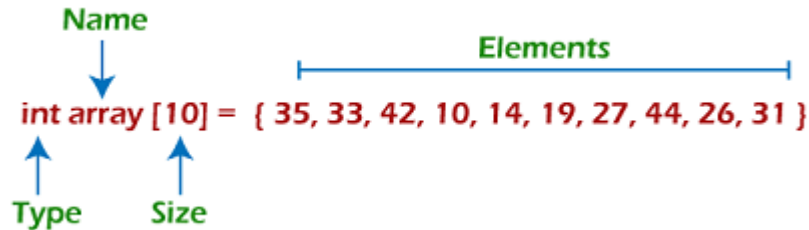
struct employee
{
    int id;
    string name;
};

int main()
{employee s1;
s1.id=1;
s1.name="amarth";
cout<<s1.id <<endl ;           //1
cout<<<<s1.name ;              //amarth

    return 0;
}
```

Array in Data Structure

- 1 Each element in an array is of the same data type and carries the same size that is 4 bytes.
- 2 Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- 3 Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.



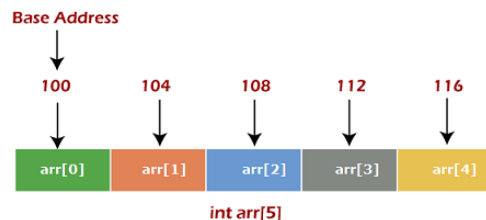
Arrays are useful because -

- 1 Sorting and searching a value in an array is easier.
- 2 Arrays are best to process multiple values quickly and easily.
- 3 **Arrays are good for storing multiple values in a single variable**

Memory allocation of an array

the data elements of an array are stored at contiguous locations in the main memory.

The name of the array represents the base address or the address of the first element in the main memory.



we have shown the memory allocation of an array `arr` of size 5.

The array follows a 0-based indexing approach.

The base address of the array is 100 bytes.

It is the address of `arr[0]`. Here, the size of the data type used is 4 bytes; therefore, each element will take 4 bytes in the

memory.

Basic operations

- 1 Traversal - This operation is used to print the elements of the array.
- 2 Insertion - It is used to add an element at a particular index.
- 3 Deletion - It is used to delete an element from a particular index.
- 4 Search - It is used to search an element using the given index or by the value.
- 5 Update - It updates an element at a particular index.

- 1 In array, space complexity for worst case is **O(n)**.
- 2 Array is homogenous. It means that the elements with similar data type can be stored in it.
- 3 In array, there is static memory allocation that is size of an array cannot be altered.
- 4 There will be wastage of memory if we store less number of elements than the declared size.

2D Array

2D array can be defined as an array of arrays.

The 2D array is organized as matrices which can be represented as the collection of rows and columns.

Syntax : `int arr [max_rows] [max_columns];`

	0	1	2	n-1
0	a[0][0]	a[0][1]	a[0][2]	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	a[2][n-1]
3	a[3][0]	a[3][1]	a[3][2]	a[3][n-1]
4	a[4][0]	a[4][1]	a[4][2]	a[4][n-1]
⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]	a[n-1][n-1]

$a[n][n]$

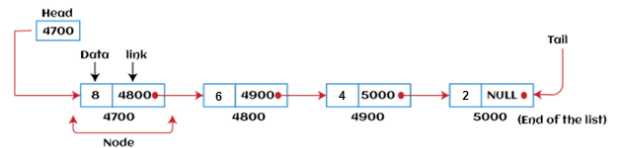
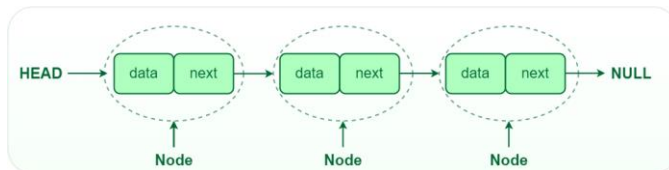
the elements are organized in the form of rows and columns.

First element of the first row is represented by $a[0][0]$ where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

Linked List

Linked List is a linear data structure, in which elements are not stored at a contiguous location, rather they are linked using pointers.

Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.



Node Structure: A node in a linked list typically consists of two components:

Data: It holds the actual value or data associated with the node.

Next Pointer: It stores the memory address (reference) of the next node in the sequence.

Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL, indicating the end of the list. This node is known as the tail node.

Why use linked list over array?

- 1 The size of the array must be known in advance before using it in the program.
- 2 Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- 3 All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

- 1 It allocates the memory dynamically.
- 2 All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- 3 In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.
- 4 Linked list is worked on heap memory
- 5 Heterogeneous - this we can store different data type in a single variable

How to declare a linked list?

Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type structure.

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

Int = 4 byte

*next = has int memory of 4 byte

we have defined a structure named as **node** that contains two variables, one is **data** that is of integer type, and another one is **next** that is a pointer which contains the address of next node.

Advantages of Linked Lists

Dynamic Size: Linked lists can grow or shrink dynamically, as memory allocation is done at runtime.

Insertion and Deletion: Adding or removing elements from a linked list is efficient, especially for large lists.

Flexibility: Linked lists can be easily reorganized and modified without requiring a contiguous block of memory.

Disadvantages of Linked Lists

Random Access: Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.

Extra Memory: Linked lists require additional memory for storing the pointers, compared to arrays.

```

#include<iostream>
using namespace std;
struct zz
{
    int data1;
    char data2;
    struct zz *link;
};

int main()
{zz obj1 ,obj2 ,obj3;

// initialization
obj1.link=NULL;
obj1.data1=10;
obj1.data2='a';
// initialization
obj2.link=NULL;
obj2.data1=30;
obj2.data2='b';

obj1.link=&obj2;           //obj1 store the address of obj2

//accessing the data member
//cout<<obj1.link->data2;  //b      obj1 se obj2 ke data2 ko access kr raha hai

// initialization
obj3.link=NULL;
obj3.data1=40;
obj3.data2='c';

obj2.link=&obj3;           //obj2 store the address of obj3

cout<<obj1.link->link->data1;  //40      obj1 se obj3 ke data1 ko access kr raha hai

return 0;
}

```

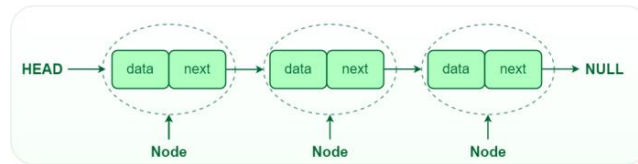
Types of linked lists:

There are mainly three types of linked lists:

- 1 Single-linked list
- 2 Double linked list
- 3 Circular singly linked list
- 4 Circular doubly linked list

1. Single-linked list:

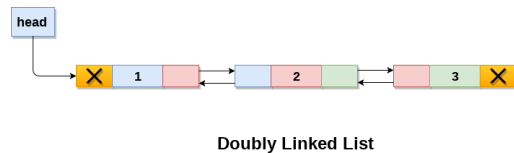
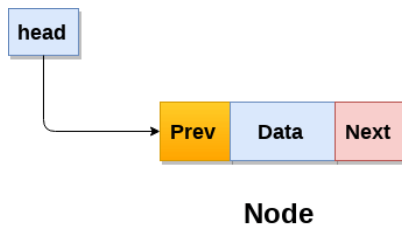
In a singly linked list, each node contains a reference to the next node in the sequence. Traversing a singly linked list is done in a forward direction.



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

2. Double-linked list:

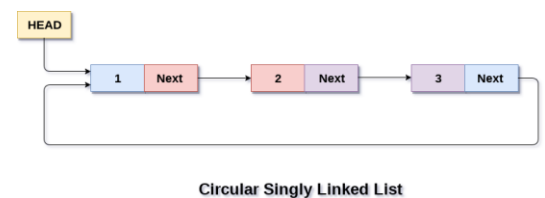
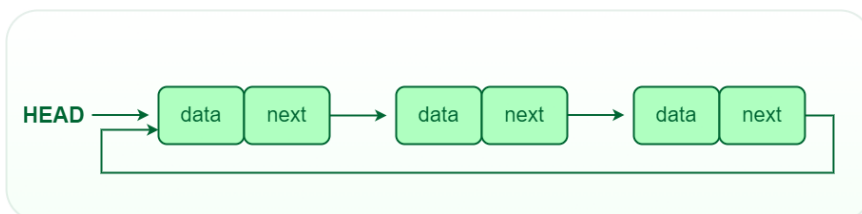
In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.



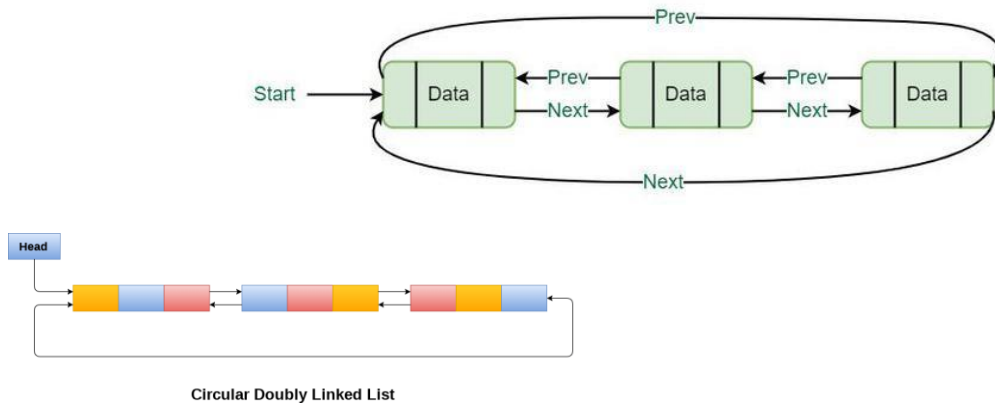
```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
}
```

3. Circular linked list:

In a circular linked list, the last node points back to the head node, creating a circular structure. It can be either singly or doubly linked.



4.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 nodes.

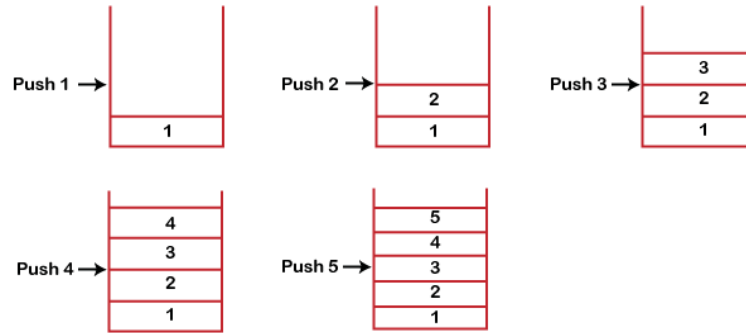
The last node of the list contains the address of the first node of the list.

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

malloc() and calloc() use in c
memory release - free() use
c++ allot memory-> new()
delete memory del()

Stack

- 1 A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.
- 2 Stack has one end, whereas the Queue has two ends (**front and rear**).
- 3 It contains only one pointer **top pointer** pointing to the topmost element of the stack.
- 4 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.
- 5 In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***
- 6 A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- 7 It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.



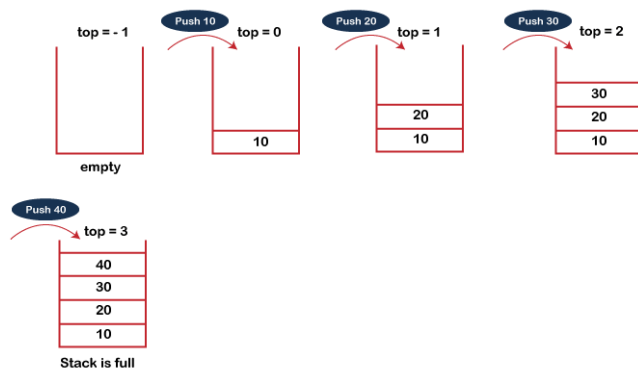
The following are some common operations implemented on the stack:

- 1 **push():** 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.
- 2 **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- 3 **isEmpty():** It determines whether the stack is empty or not.
- 4 **isFull():** It determines whether the stack is full or not.'
- 5 **peek():** It returns the element at the given position.
- 6 **count():** It returns the total number of elements available in a stack.
- 7 **change():** It changes the element at the given position.
- 8 **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

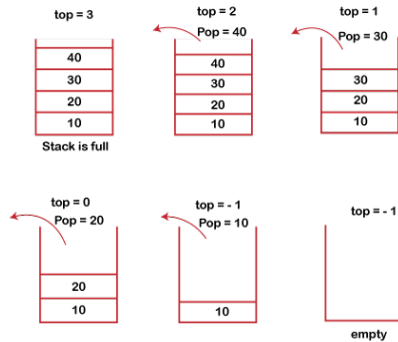
- 1 Before inserting an element in a stack, we check whether the stack is full.
- 2 If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.
- 3 When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- 4 When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1**, and the element will be placed at the new position of the **top**.
- 5 The elements will be inserted until we reach the **max** size of the stack.



POP operation

The steps involved in the POP operation is given below:

- 1 Before deleting the element from the stack, we check whether the stack is empty.
- 2 If we try to delete the element from the empty stack, then the **underflow** condition occurs.
- 3 If the stack is not empty, we first access the element which is pointed by the **top**
- 4 Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Applications of Stack

- 1 Balancing of symbols:
- 2 String reversal:
- 3 UNDO/REDO:
- 4 DFS(Depth First Search):
- 5 Expression conversion:
- 6 Recursion:

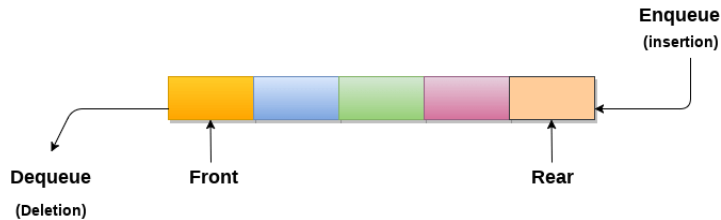
```
#include<iostream>
#include<stack>           //it is a template
using namespace std;
int main()
{
    stack<int> s;
    for (int i = 1; i <=10; i++)
        {s.push(i);}
    while(!s.empty())    //this loop untill the s is empty
    {
        cout<<s.top() <<"\t";
        //10      9      8      7      6      5      4      3      2      1
        s.pop();
    }
    return 0;
}
```

Queue

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

Queue is referred to be as First In First Out list.

For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

- 1 Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.
- 2 Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- 3 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.
- 4 Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
- 5 Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
- 6 Queues are used in operating systems for handling interrupts.

Complexity

Data Structure	Time Complexity								Space Compleity	
	Average				Worst					Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	

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

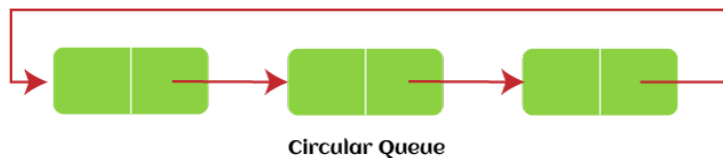
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



- 1 The major drawback of using a linear Queue is that insertion is done only from the rear end.
- 2 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.
- 3 In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -

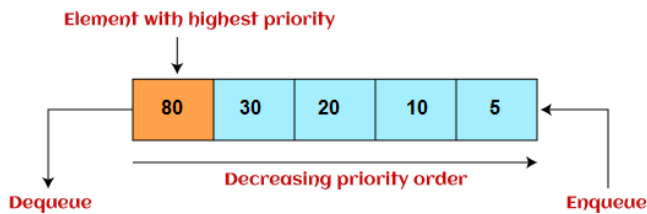


Priority Queue

It is a special type of queue in which the elements are arranged based on the priority.

It is a special type of queue data structure in which every element has a priority associated with it.

Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle.



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms

There are two types of priority queue that are discussed as follows -

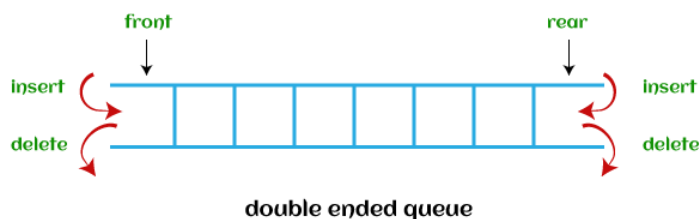
Ascending priority queue - In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.

Descending priority queue - In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Deque (or, Double Ended Queue)

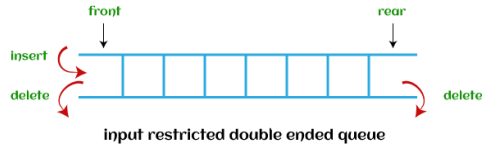
In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear.

It means that we can insert and delete elements from both front and rear ends of the queue. Deque can b

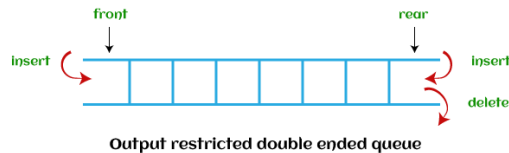


There are two types of deque that are discussed as follows –

Input restricted deque - As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



Output restricted deque - As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

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

Dequeue: It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.

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

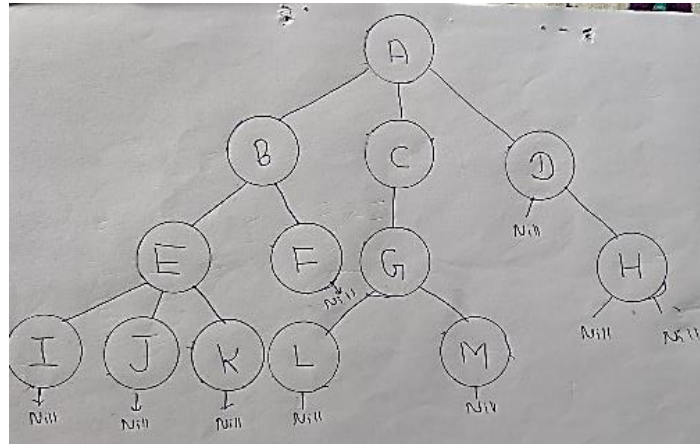
Queue overflow (isfull): It shows the overflow condition when the queue is completely full.

Queue underflow (isempty): It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

```
//queue = pop,push,front,empty,size, function
#include<iostream>
#include<queue>           //it is a template
using namespace std;
int main()
{ queue<int> s;
  for (int i = 1; i <=10; i++)
    {s.push(i);}
  while(!s.empty())           //this loop untill the s is empty
  { cout<<s.front() <<"\t";   //1 2 3 4 5 6 7 8 9 10
    s.pop();
  }
  return 0;}
```

Tree

1. A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
2. A tree data structure is a non-linear data structure because it does not store in a sequential manner.
3. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- 4.
5. In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type.



ROOT=A

NODES=A,B,C,D,E,F,G,H,I,J,K,L,M,N

PARENT NODE=Parent node is predecessor of child node

CHILD NODE: child node is the successor of Parent node

LEAF NODE(external node):A node which has no child node(I,J,K,L,M,F,H)

NON-LEAF NODE(internal node) (A ,B,C,D,E,G):A node contain at least one leaf node.

DEPTH OF NODE: length of path from ROOT to that node

HEIGHT OF NODE:Number of edges in the longest path from that node to leaf.

SIBLING: Children of same parent

DEGREE: Degree of node is number of children of that node OR degree of tree=max degree among all nodes

LEVEL=HEIGHT

PATH: sequence of consecutive edges from source node to destination node

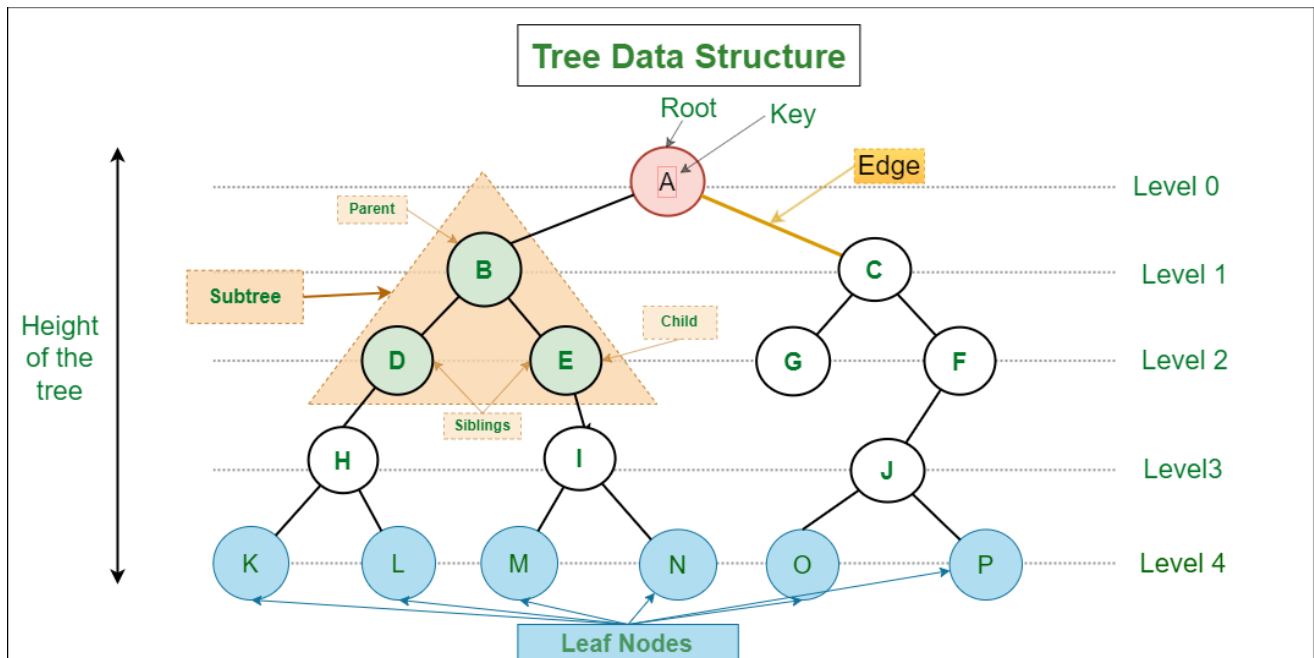
EDGE: link between two nodes

Ancestor: A->B HERE A IS ANCESTOR: Any predecessor node from ROOT to edge L=A,C,G

Descendent: Any successor node on the path from that node to leaf node.

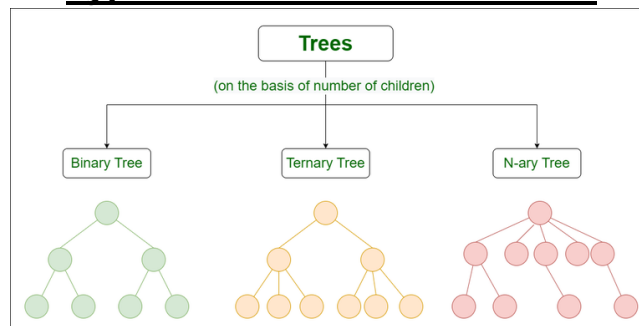
Ex : C = G,L,M Ex: B = F,E,I,J,K

SUBTREE: It is containing node of its descendent .



- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {K, L, M, N, O, P} are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}
- **Descendant:** Any successor node on the path from the leaf node to that node. {E,I} are the descendants of the node {B}.
- **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree:** Any node of the tree along with its descendant.

Types of Tree data structures:



Binary tree:

In a binary tree, each node can have a maximum of two children linked to it.

Types of binary tree

- 1.full binary trees
- 2.complete binary trees
- 3.balanced binary trees
- 4.degenerate or pathological binary trees.

Ternary Tree:

A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.

N-ary Tree or Generic Tree:

Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed).

Unlike the linked list, each node stores the address of multiple nodes.

Basic Operation Of Tree Data Structure:

Create – create a tree in the data structure.

Insert – Inserts data in a tree.

Search – Searches specific data in a tree to check whether it is present or not.

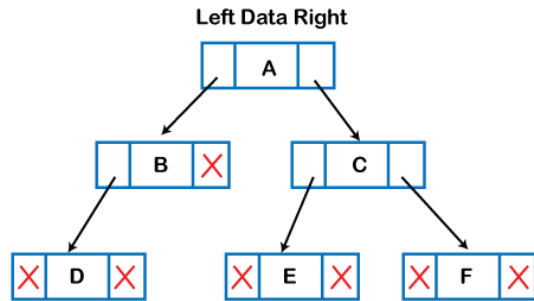
Traversal (travel):

1. **Preorder Traversal** – perform Traveling a tree in a pre-order manner. Root->Left->Right(First)
2. **In order Traversal** – perform Traveling a tree in an in-order manner. Left->Root->Right(Second)
3. **Post-order Traversal** –perform Traveling a tree in a post-order manner. Left->Right->Root(Third)

Appication of Tree :FILE SYSTEM,ROUTING PROTOCOL etc.

Implementation of Tree

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



The above figure shows the representation of the tree data structure in the memory.

In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

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

BINARY TREE:

1. Each node have Maximum two children(0,1,2)
2. Max nodes possible at any level: 2^i

Types Of Binary Tree:

1.Proper/Strict Binary tree: (Each node will contain exactly 2 children except leaf node)

Ex: no of leaf node=no of internal node+1

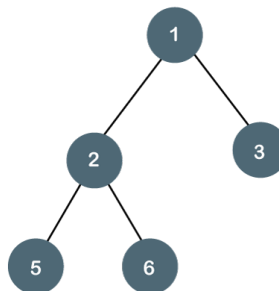
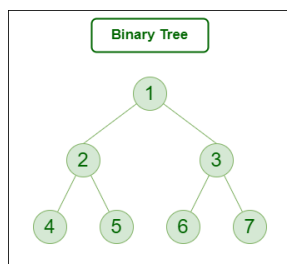
2.Complete Binary Tree: All the level are completely filled node followed by left possible .

3.FULL BINARY Tree: All internal nodes should contain 2 children and all leaf at same source level

4.DEGENARATE Tree: All internal nodes have only one child.

Types:

1. Left skewed
2. Right Skewed



Tree Traversal Techniques

A Tree Data Structure can be traversed in following ways:

Depth First Search or DFS

1. Pre-order Traversal Root->Left->Right (1 time count)
2. In-order Traversal Left->Root->Right (2 time count)
3. Post-order Traversal Left->Right->Root (3 time count)

Level Order Traversal or Breadth First Search or BFS

1. Boundary Traversal
2. Diagonal Traversal

BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.

BFS follows Queue.

DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.

DFS follows Stack.

Inorder Traversal - Left->Root->Right (2 time count)

1. Traverse the left subtree, i.e., call Inorder(left->subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right->subtree)

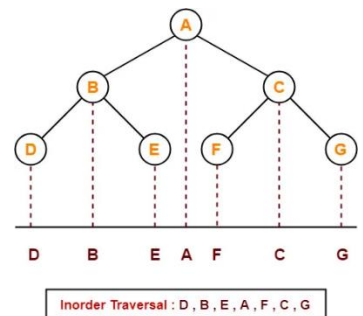
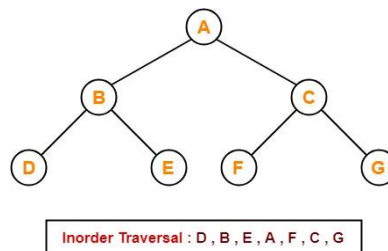
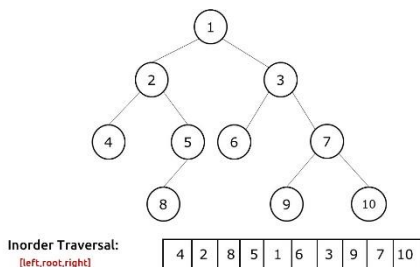
Uses of Inorder Traversal:

In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.

To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

Time Complexity: $O(N)$

Auxiliary Space: If we don't consider the size of the stack for function calls then $O(1)$ otherwise $O(h)$ where h is the height of the tree.

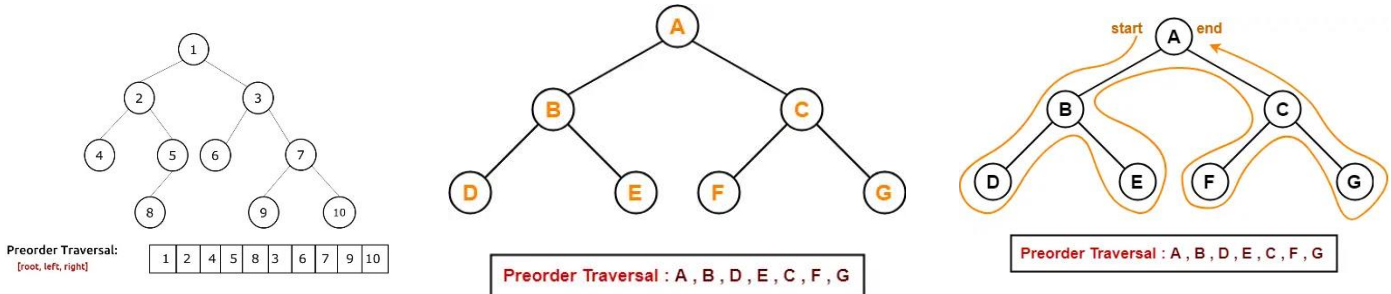


Preorder Traversal - Root->Left->Right (1 time count)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left->subtree)
3. Traverse the right subtree, i.e., call Preorder(right->subtree)

Uses of Preorder:

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.



Applications-

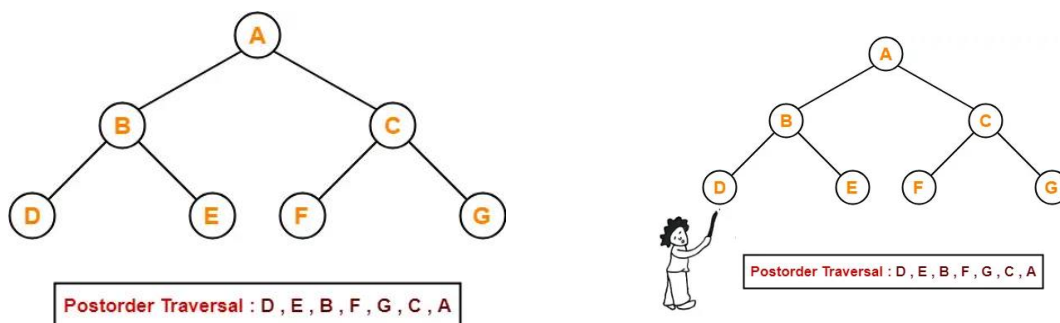
1. Preorder traversal is used to get prefix expression of an expression tree.
2. Preorder traversal is used to create a copy of the tree.

Postorder Traversal- Left → Right → Root

Traverse the left sub tree i.e. call Postorder (left sub tree)

Traverse the right sub tree i.e. call Postorder (right sub tree)

Visit the root



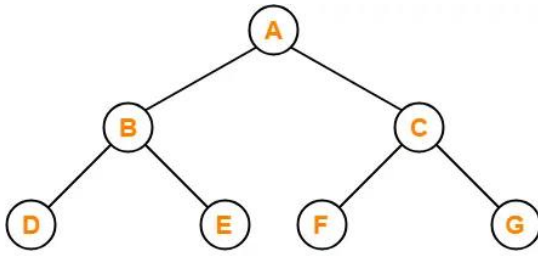
Applications-

1. Postorder traversal is used to get postfix expression of an expression tree.
2. Postorder traversal is used to delete the tree.
3. This is because it deletes the children first and then it deletes the parent.

Breadth First Traversal

Breadth First Traversal of a tree prints all the nodes of a tree level by level.

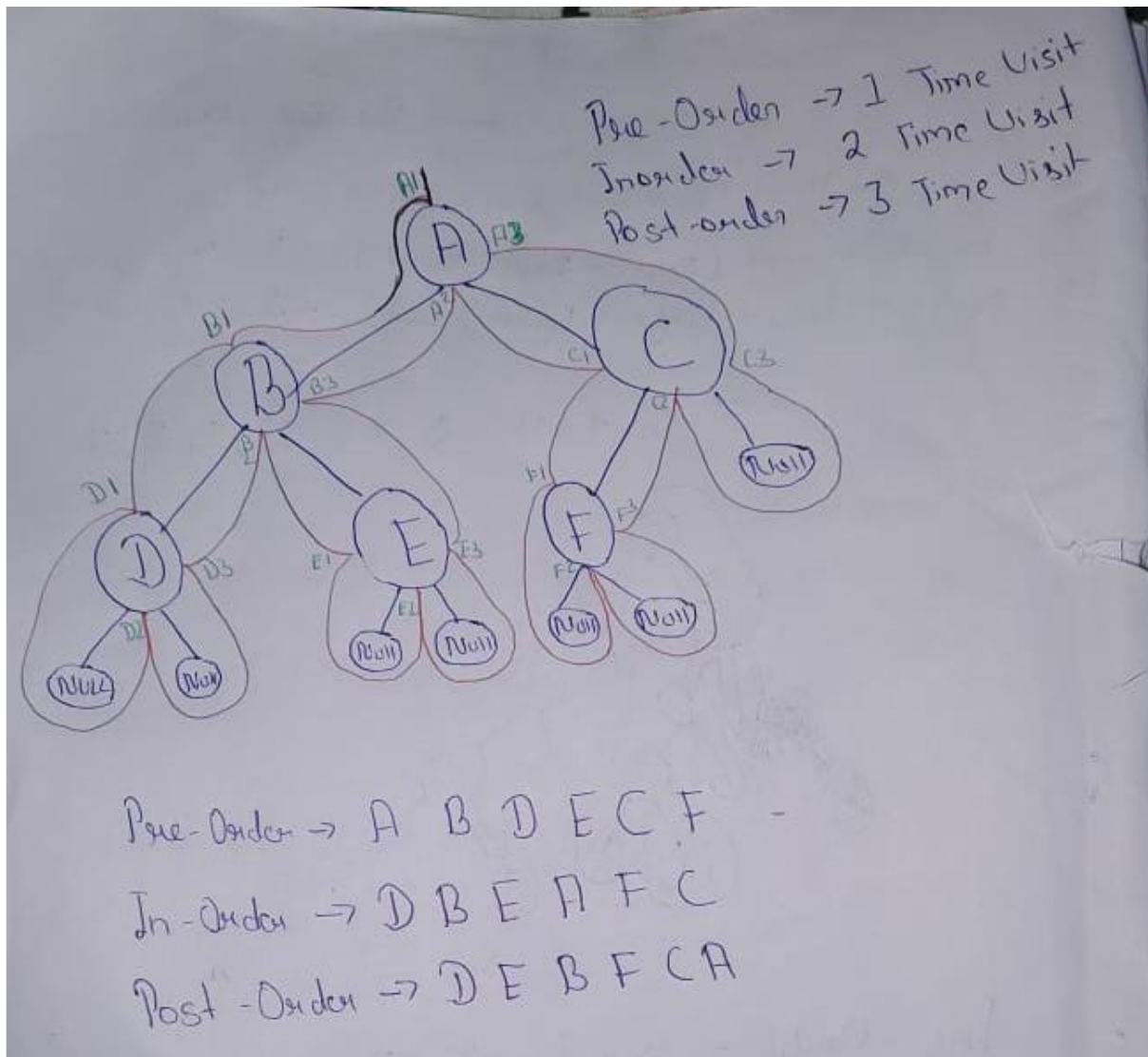
Breadth First Traversal is also called as **Level Order Traversal**.



Level Order Traversal : A, B, C, D, E, F, G

Application-

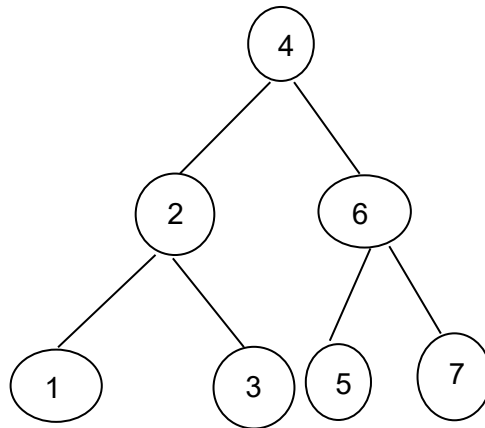
Level order traversal is used to print the data in the same order as stored in the array representation of a complete binary tree.



Binary Search Tree(BST):-

1. Binary tree → binary search tree means BST hone k liye binary tree hona jaruri hai .
2. most 2 nodes(0,1,2)
3. Left node value should be less than Root
4. Right node value should be greater than Root
5. In Equal condition either we take left node that is called left bias node or right node that is called right bias node
6. In BST ,we always get the Inorder tree as ascending order & least value will be shifted towards left node and max value towards right node

Ex: 4,2,3,6,5,7,1



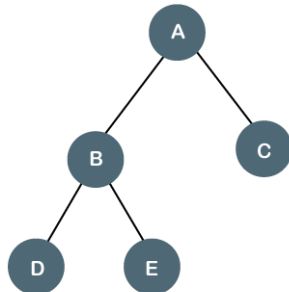
Inorder:1,2,3,4,5,6,7

Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree.

The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children.

The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

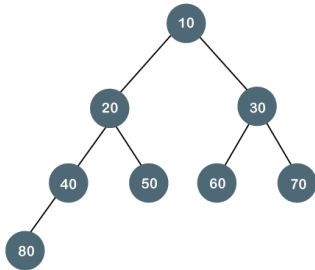
Properties of Full Binary Tree

The number of leaf nodes is equal to the number of internal nodes plus 1.

In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.

Complete Binary Tree

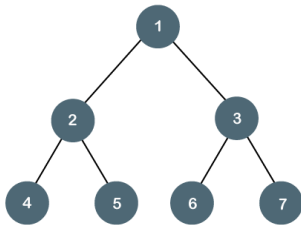
The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.



The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.



The above tree is a degenerate binary tree because all the nodes have only one child.

It is also known as a **right-skewed tree** as all the nodes have a right child only.

The above tree is also a degenerate binary tree because all the nodes have only one child.



It is also known as a **left-skewed tree** as all the nodes have a left child only.

AVL TREEE (BALANCED BST)

An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper

It can be defined as height balanced binary search tree in which each node is associated with **balance factor** which is calculated by subtracting the height of its right sub-tree from that of its left sub tree.

Tree is said to be balanced if balance factor of each node is in **between -1 to 1**, otherwise , the tree will be unbalanced and need to be balanced.

Balance Factor = Left subtree-Right subtree

To get balance the tree, we need rotation.

AVL ROTATION:

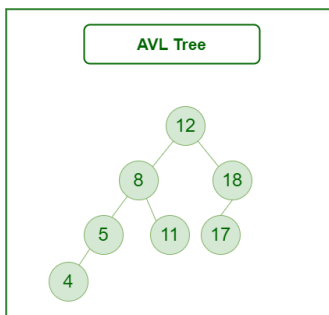
- a. LL: LEFT-LEFT
- b. RR: RIGHT-RIGHT
- c. LR: Left than Right
- d. RL: Right than Left

NOTE: Only BST can be converted into AVL

BT-Binary Tree has no order

BST-Binary Search Tree has order (left<ROOT>right)

Pattern : BT->BST->AVL



Applications of AVL Tree:

It is used to index huge records in a database and also to efficiently search in that.

For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.

Database applications, where insertions and deletions are less common but frequent data lookups are necessary

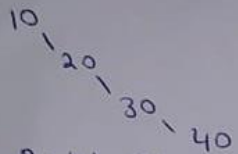
Software that needs optimized search.

It is applied in corporate areas and storyline games.



AVL $\rightarrow (\log n)$

10, 20, 30, 40

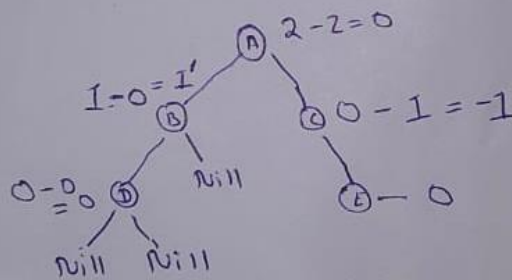


\leftarrow So Comp is $O(n)$.

Right Skewed.

- ① To solve this problem, AVL Tree comes with the help of Balance factors.

BF = height of left - height of Right

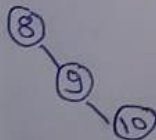


\rightarrow AVL Tree.

Between, -1, 0, +1

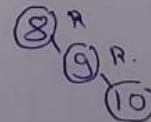
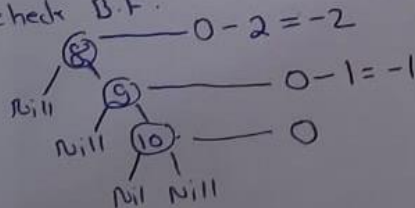
If not Balanced Do rotations

① 8, 9, 10



- This is not BST we have to Balance it.

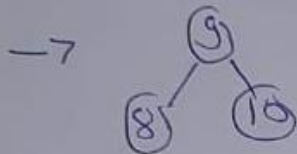
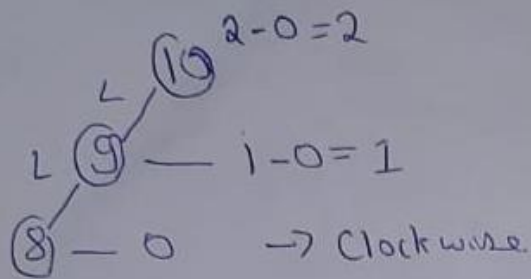
- check B.F.



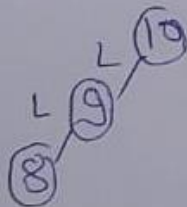
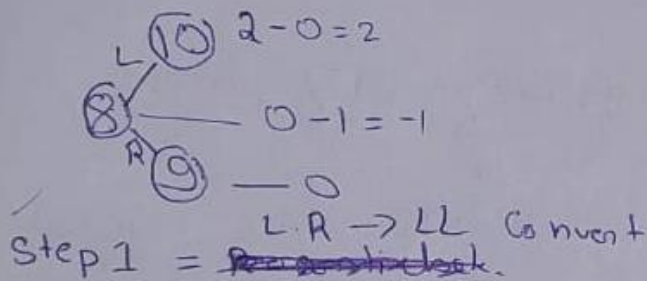
RR - anti-clock



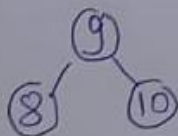
$\Rightarrow 10, 9, 8$



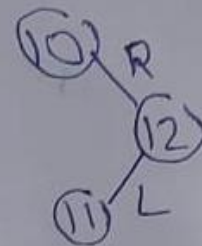
L.R.
 $= 10, 8, 9$



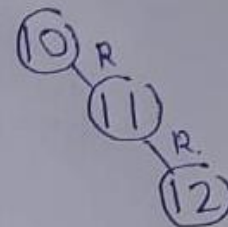
Step 2 \rightarrow L - clock.



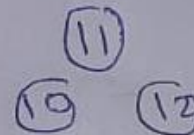
$\Rightarrow 10, 12, 11$



Step 1 = RL \rightarrow RR

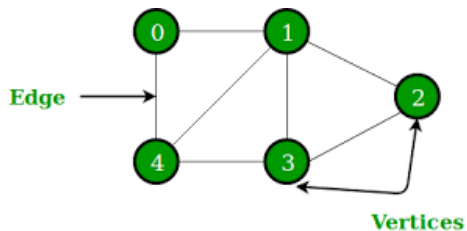


Step 2 \rightarrow anti clock



Graph:

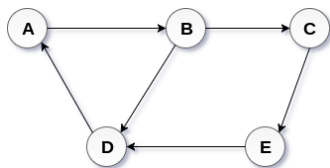
1. It is Nonlinear
2. It is a collection of vertices and edges $G=\{V,E\}$
3. It is no unique node
4. It is a Network Model
5. It is cyclic(loop can be performed)
6. It contains no of edges
7. Application :Finding shortest path of networking(google map ,communication ,Bio etc.)



GRAPHS ARE TWO TYPE

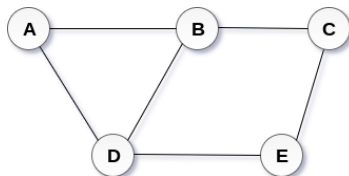
1. Direct graph
2. Undirect graph

Direct Graph In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.



Directed Graph

Undirected graph in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.



Undirected Graph

IN THIS EXAMPLE

$|V|$ =order of graph=total no of vertex = 5

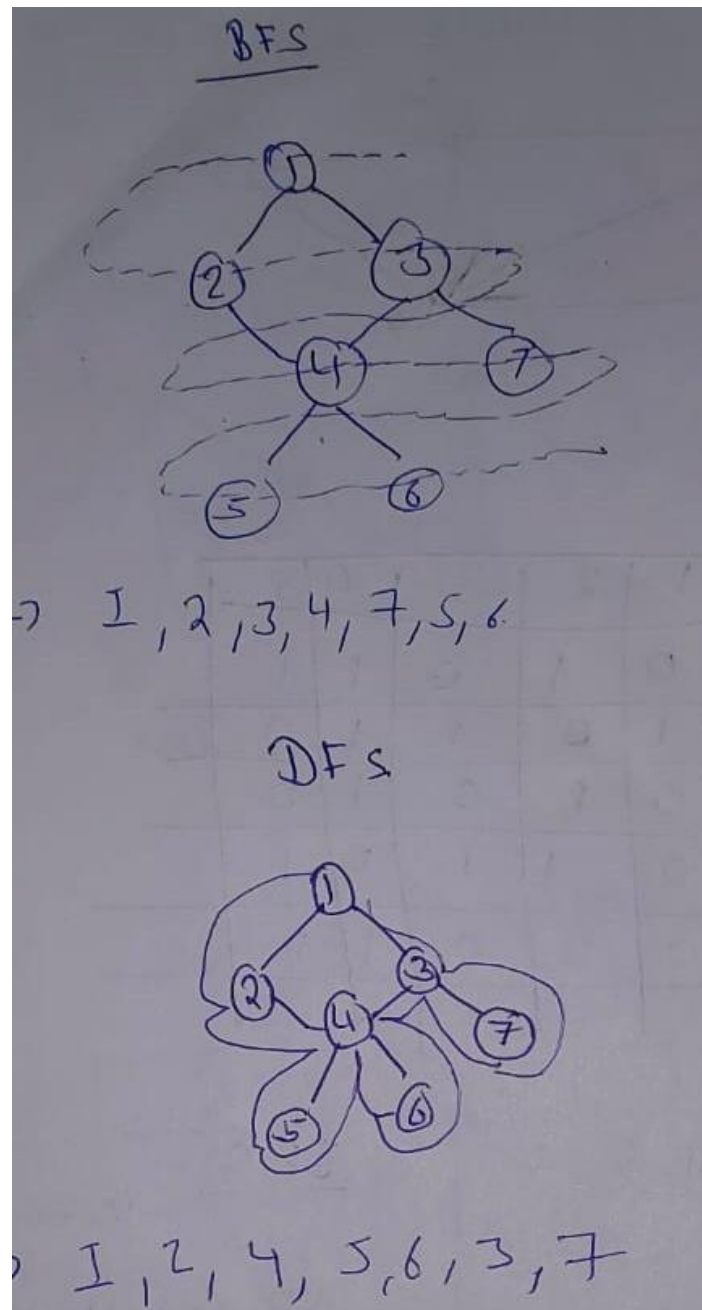
$|E|$ =size of graph=total no of edges = 6

BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.

BFS follows Queue.

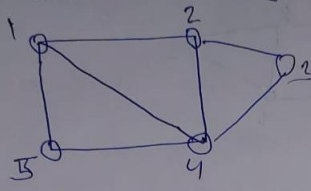
DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.

DFS follows Stack.



= Adjacent Matrix
~~n x n~~ $n \times n = O(n^2)$

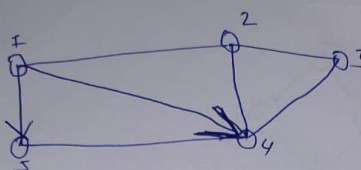
Graph of 5 nodes using array



order of vertices = 5.

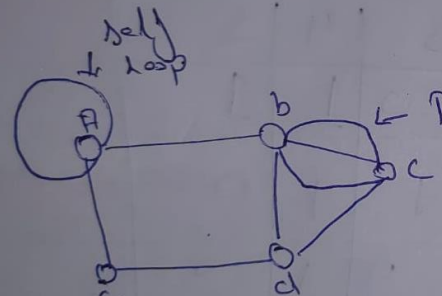
$n \times n = 5 \times 5$
 = 5 row, 5 column.

	1	2	3	4	5
R. 1	0	1	0	1	1
R. 2	1	0	1	1	0
R. 3	0	1	0	1	0
R. 4	1	1	1	0	1
R. 5	1	0	0	1	0



	1	2	3	4	5
1	0	1	0	1	1
2	1	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	0	0	0	1	0

① Self loop ② parallel loop



Self loop

parallel loop