

# Data Structures and Algorithms (BCSE202L)

*Module 1: Algorithm Analysis*

Dr. Priyanka N

# What is data?

## Definition (Dictionary):

- The quantities, characters, or symbols on which operations are performed by a computer, which may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media

## Example:

$$C=A * B + C$$

**C,A,B,C** → Data

\* → Multiplication Operation

+ → Addition Operation

# Data and Information

Example:

*PIKANAMYAIR*

**PIK ANA MYAIR**

→ **Data**

**I AM PRIYANKA**

→ **Information**

- If data is arranged systematically, it gets a structure and becomes meaningful

# Data structures

A  
B  
C  
D  
E  
F

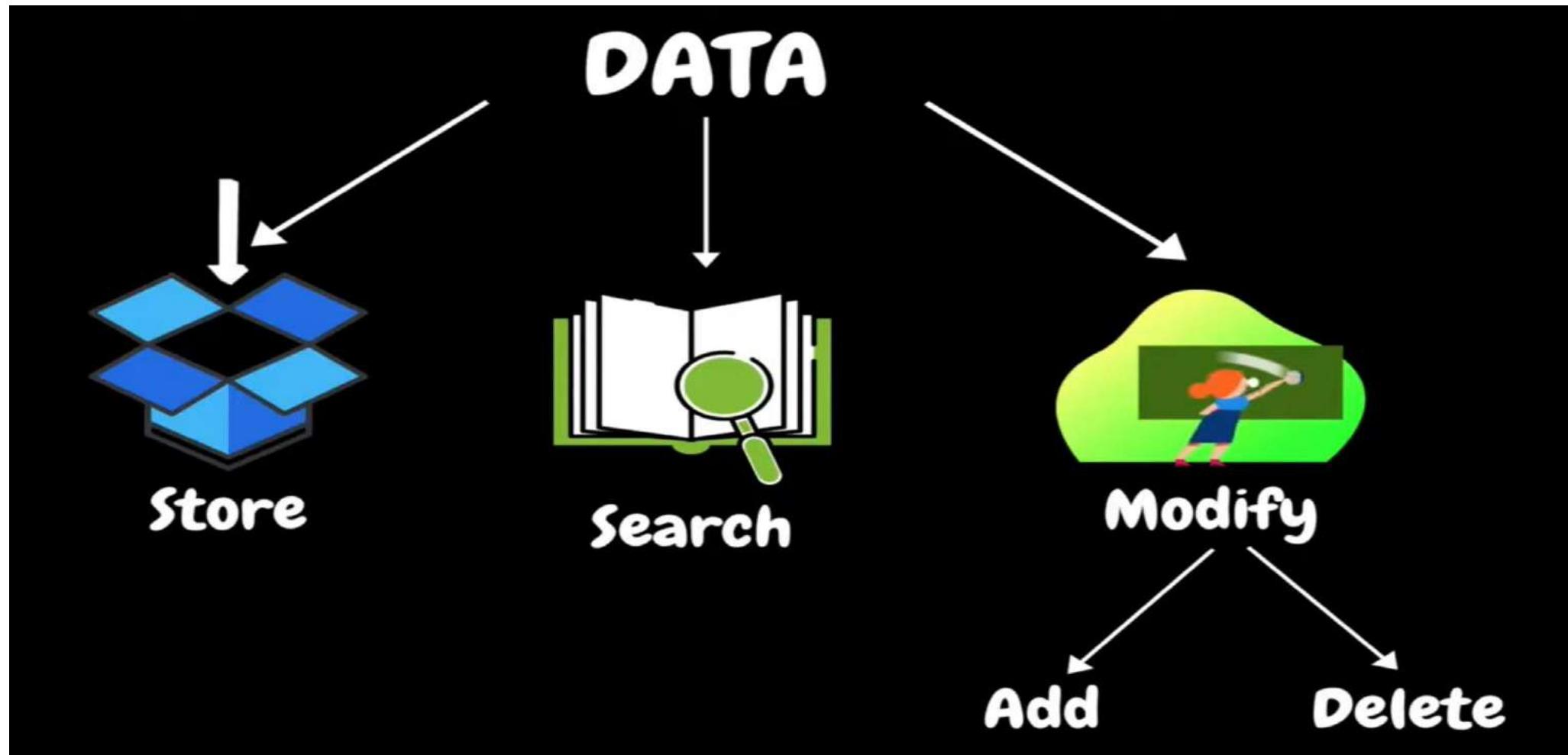


# Data structures

Fiction  
Tech  
Classic  
Kids  
History  
Romance

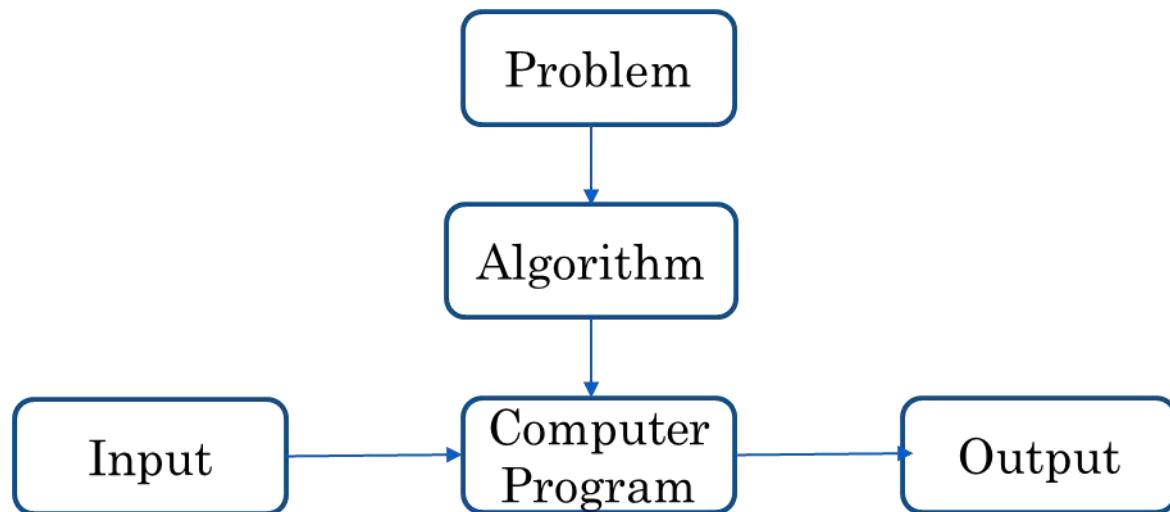


# Data structures



# Algorithms

- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output
- An algorithm is also considered a tool for solving a well-specified computational problem



# Example

## Problem:

Design an algorithm to add two numbers and display the result

### ALGORITHM

**Step 1** – START

**Step 2** – declare three variables A, B & C

**Step 3** – define values of A & B

**Step 4** – ADD values of A & B

**Step 5** – STORE output of step 4 to C

**Step 6** – PRINT C

**Step 7** – STOP

### PROGRAM CODE

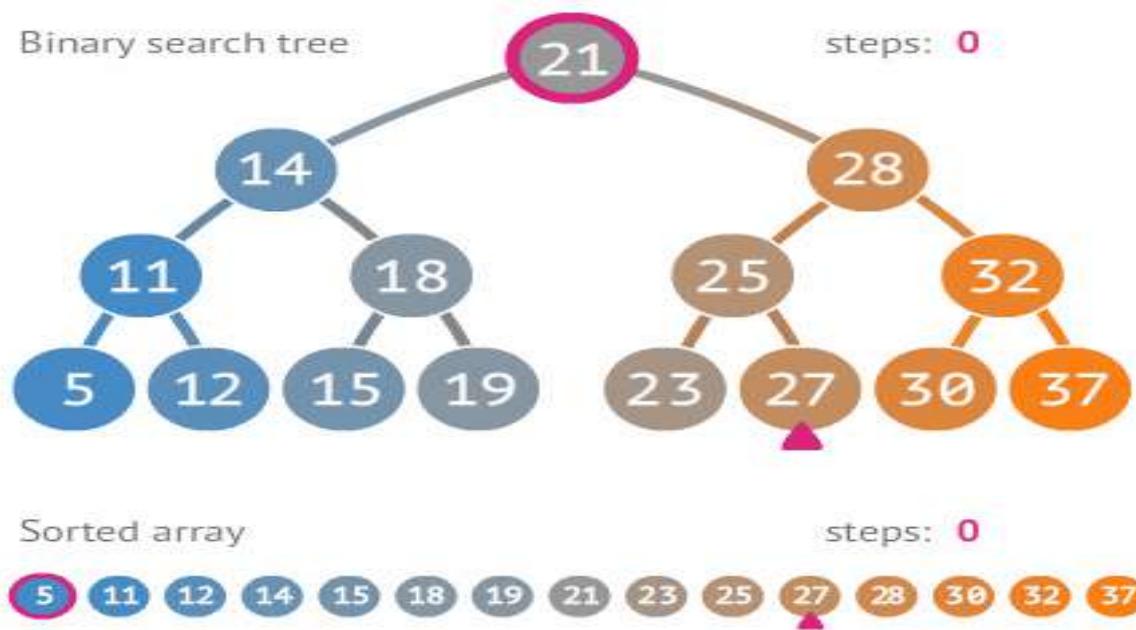
```
#include <iostream>
using namespace std;
int main()
{
    int A,B,C;
    cout<<"Enter values for two numbers:\n";
    cin>>A>>B;
    C=A+B;
    cout<<"The addition of two numbers is:"<<C;
    return 0;
}
```

# Examples of Real-World Algorithms

- Sorting Papers (Imagine a teacher sorting their students' papers according to the alphabetical order of their first names)
- Facial Recognition
- Google Search
- Traffic Lights
- Bus Schedules

# Algorithms and data structures

- Data structures and algorithms are the building blocks of coding
- Data structures allow us to organize and store data, while algorithms allow us to process that data in a meaningful way



*Data structure and algorithms helps to understand the nature of the problem at a deeper level and thereby better understand the world*

# Importance of algorithms and data structures

- Comprehensive knowledge of data structures in combination with algorithms is the core foundation of writing good codes. It reduces coding costs and enhances data accuracy, which is the ultimate goal of organizations
- To earn salaries as high as the developers of Amazon and Google, you need to improve your problem-solving abilities by mastering data structures and algorithms
- Learning data structures and algorithms is also beneficial for a better understanding of new frameworks such as Angular, React, Vue, Spring MVC, etc
- When you get exposure to a different range of problem-solving techniques, it helps you take up the next challenging problem easily
- It can be used to determine how a problem is represented internally, how the actual storage pattern works & what is happening under the hood for a problem

# Data Structures and Algorithms (BCSE202L)

*Module 1: Algorithm Analysis*

Dr. Priyanka N

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		
<b>Module:8</b>	<b>Contemporary Issues</b>	<b>2 hours</b>

# Time complexity

- The amount of time taken by an algorithm to execute each statement of code of an algorithm till its completion with respect to the function of the length of the input
- The time complexity of algorithms is most commonly expressed using the *big O* notation which is an asymptotic notation to represent the time complexity

# Algorithm Efficiency- Performance Analysis

- Based on Time and Space Complexity
  - Best Case( Min Amount of Time)
  - Worst Case( Max Amount of Time)
  - Average Case( Average amount of Time)

# Time Complexity

- The amount of time taken by an algorithm to run for completion
- It is measured by number of primitive operations required to produce output
- Time complexity is independent of
  - Speed
  - Word length
  - OS
  - Translation Time

# Space complexity

- The amount of space required by an algorithm to run on completion
- Mostly, time and space complexity is computed as a function of input size

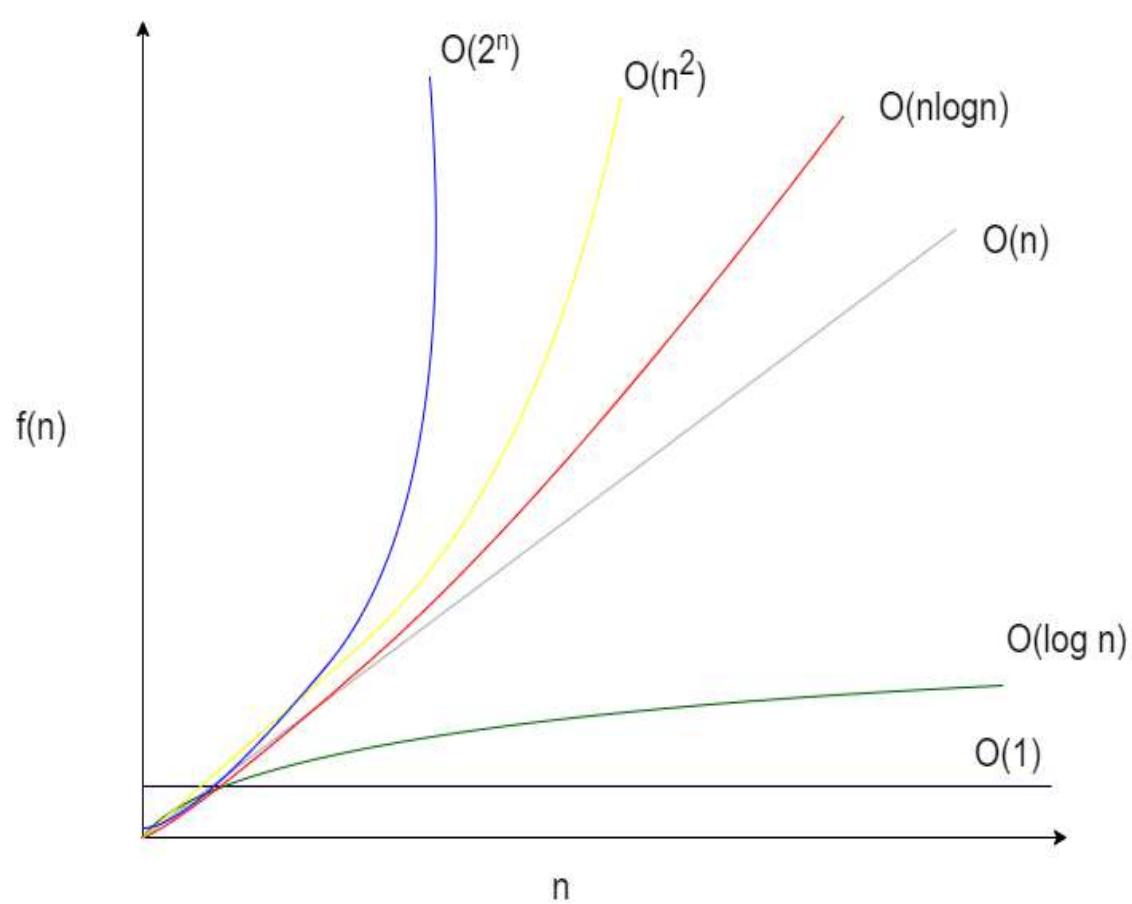
# Calculation of Time Complexity

- Two Approaches
  - Frequency Count or Step Count
  - Asymptotic Notions

# Frequency Count/Step Count Method

- Number of times a statement is executed
- RULES:
  - For comments, and declarations, STEP COUNT=0
  - For return, assignment statements, STEP COUNT=1
  - Ignore lower-order exponents when higher-order exponents are present
  - Ignore constant multipliers

# Time complexity



$O(N!)$	Factorial
$O(2^N)$	Exponential
$O(N^3)$	Cubic
$O(N^2)$	Quadratic
$O(N \log N)$	$N \times \log N$
$O(N)$	Linear
$O(\log N)$	Logarithmic
$O(1)$	Constant

# Time complexity - O(1)

- Where an algorithm's execution time is not based on the input size  $n$ , it is said to have constant time complexity with order  $O(1)$
- Whatever input size  $n$ , the runtime doesn't change

```
#include <bits/stdc++.h>
using namespace std;
int main( ){
    cout<<"Hello World!!";
}
```

# Time complexity - O(1)

- Although there are separate statements inside the main function, the overall time complexity is still O(1) because each statement takes only one unit of time for overall execution

```
#include<bits/stdc++.h>
using namespace std;
int main( ){
    cout<<"Hello dev 1\n";
    cout<<"Hello dev 2\n";
    cout<<"Hello dev 3\n";
}
```

# Time complexity - O(1)

- In the above code “Hello World” is printed only once on the screen
- The time complexity is O(1)

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

int main()
{
    cout << "Hello World";
    return 0;
}
```

# What is log?

$$2^4 = 16$$

A diagram illustrating the inverse relationship between exponentiation and logarithms. At the top, the equation  $2^4 = 16$  is shown. Two arrows point downwards from the base 2 and the exponent 4 to the logarithmic expression  $\log_2 16 = 4$  below.

$$\log_2 16 = 4$$

**log IN MATHEMATIS**  $\longrightarrow \log_{10}$

$$\log_2 8 = 3$$

**log IN COMPUTER SCIENCE**  $\longrightarrow \log_2$

$$\log_2 1024 = 10$$

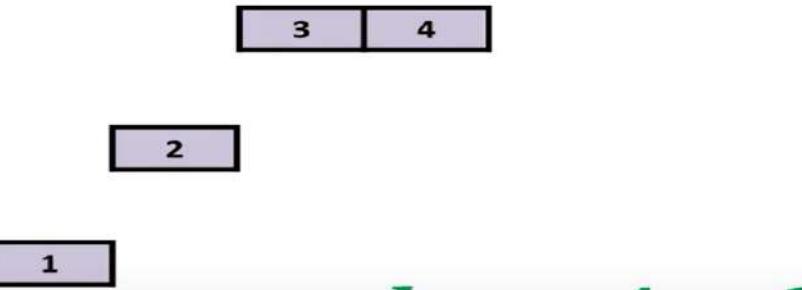
$$\log_2 12 = 3.5$$

$$\log_3 27 = 3$$

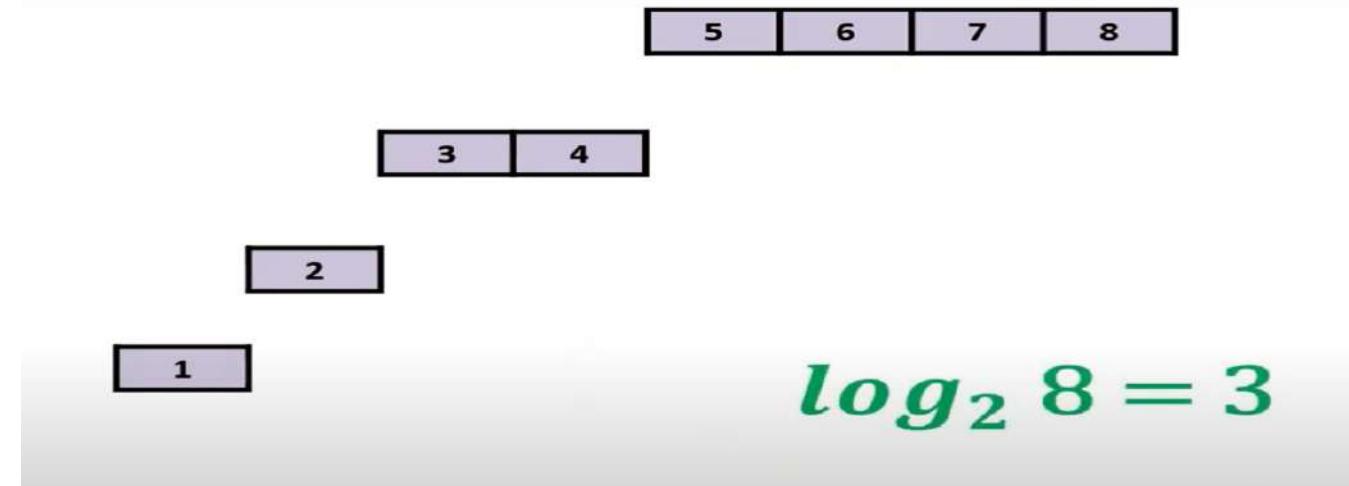
# Time complexity – O (log n)

- When an algorithm decreases the magnitude of the input data in each step, it is said to have a logarithmic time complexity
- $O(\log n)$  basically implies that time increases linearly while the value of 'n'; increases exponentially
- Example: If computing 10 elements take 1 second, computing 100 elements take 2 seconds, computing 1000 elements take 3 seconds, and so on
- When using divide and conquer algorithms, such as binary search, the time complexity is  $O(\log n)$

# Time complexity – O (log n)



$$\log_2 4 = 2$$



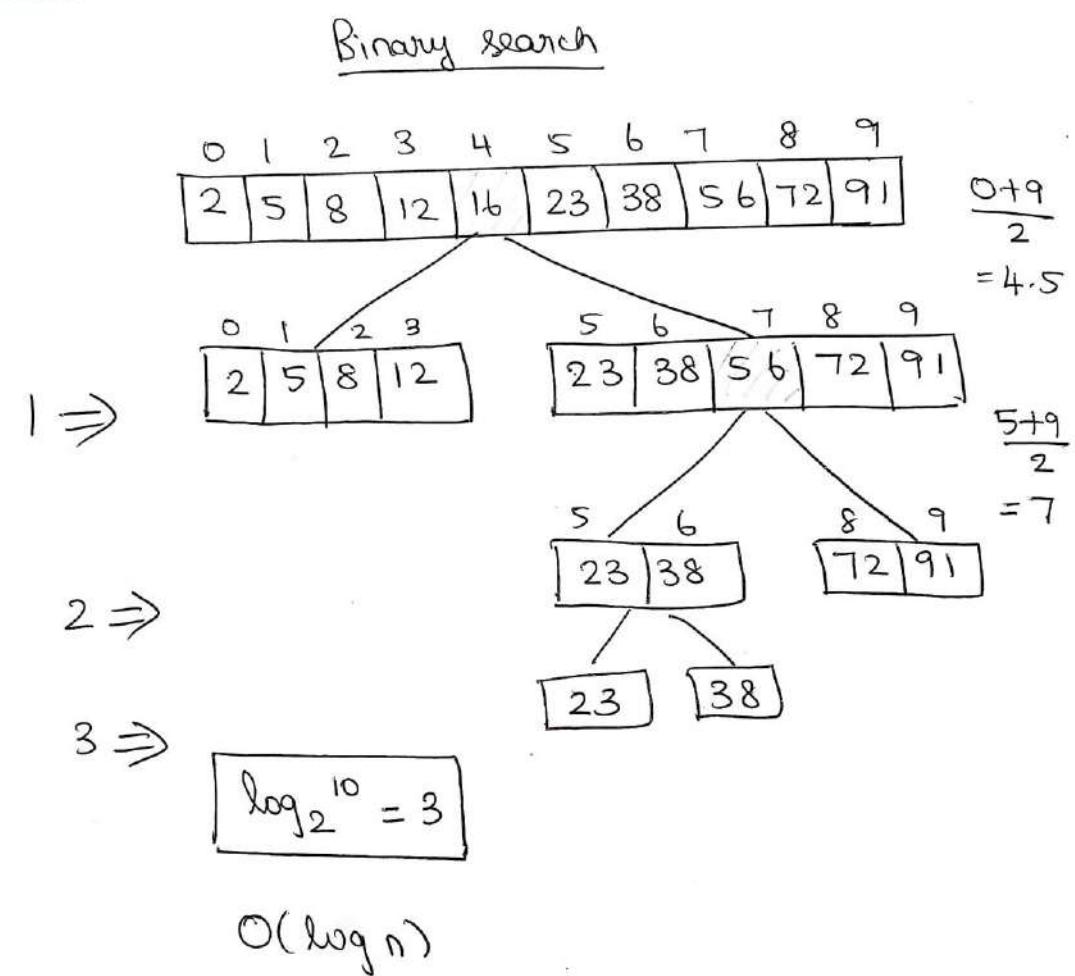
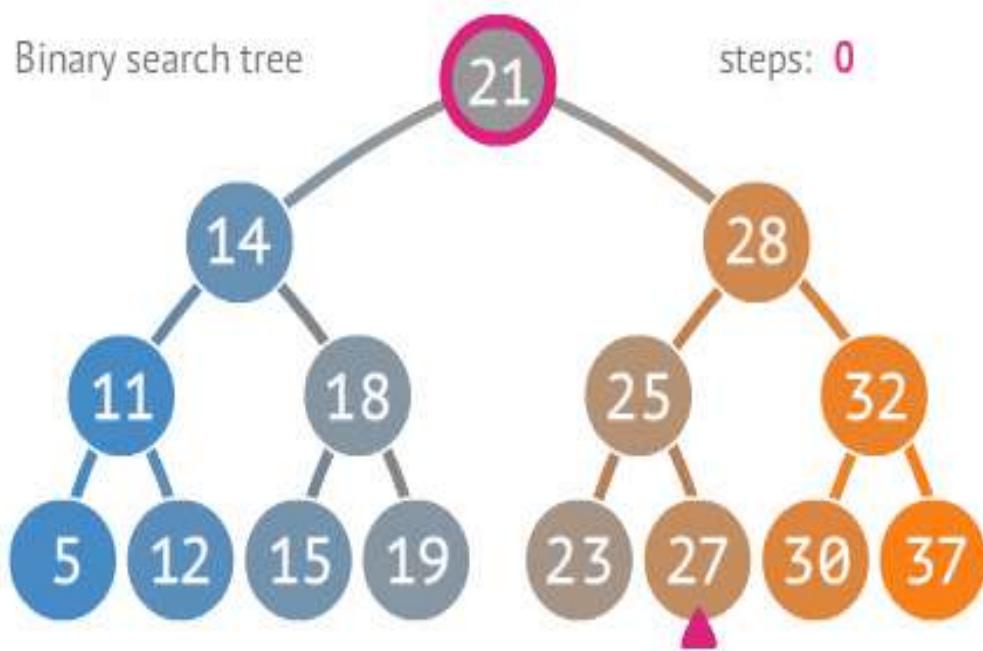
$$\log_2 8 = 3$$

**Array length = 4 >>2       $\log_2 4 = 2$**

**Array length = 8 >>3       $\log_2 8 = 3$**

**TIME COMPLEXITY : O(log(n))**

# Time complexity – O (log n)



# Time complexity – O (log n)

```
int binarySearch(int arr[], int l, int r, int x){  
    if (r >= l) {  
        int mid = l + (r - l) / 2;  
        if (arr[mid] == x)  
            return mid;  
        if (arr[mid] > x)  
            return binarySearch(arr, l, mid - 1, x);  
        return binarySearch(arr, mid + 1, r, x);  
    }  
    return -1;  
}
```

# Time complexity – O (log n)

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

int main()
{
    int i, n = 8;
    for (i = 1; i <= n; i=i*2) {
        cout << "Hello World !!!\n";
    }
    return 0;
}
```

n=8			
	i value	condition	Output
	1	$1 \leq 8$ → Condition is True	Hello world!!!
	2	$2 \leq 8$ → True	Hello world!!! Hello world!!!
	4	$4 \leq 8$ → True	Hello world!!! Hello world!!! Hello world!!!
	8	$8 \leq 8$ → True	Hello world!!! Hello world!!! Hello world!!! Hello world!!!
	16	$16 \leq 8$ → False	Exit

- When an algorithm decreases the magnitude of the input data in each step, it is said to have a logarithmic time complexity of O (log n)

# Time complexity - O(n)

- When the running time of an algorithm increases linearly with the length of the input, it is assumed to have linear time complexity of order O(n)

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n;
    cin>>n;
    for(int i=0;i<n;i++){
        cout<<"Hello fellow Developer!!\n";
    }
}
```

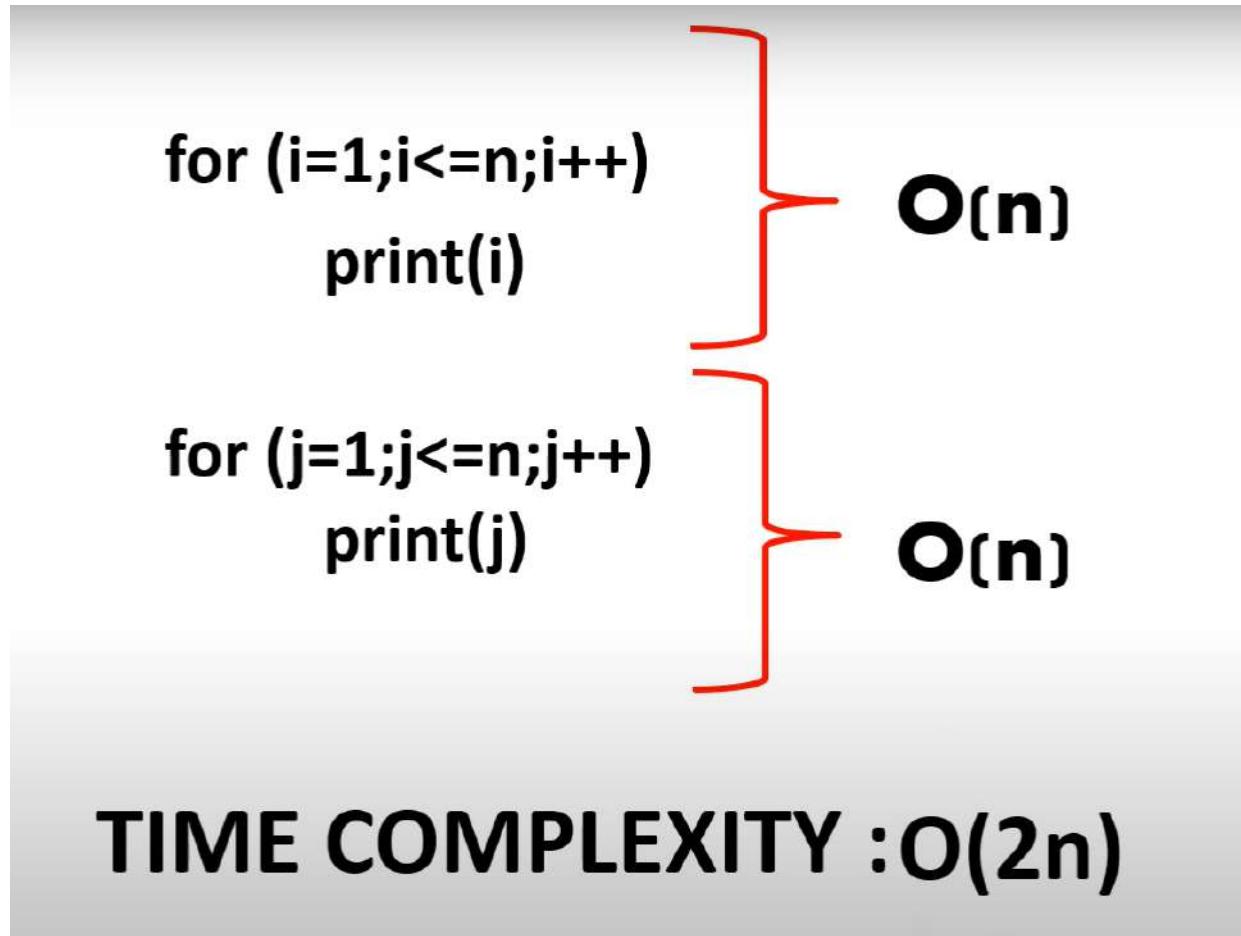
```
5
Hello fellow Dev!!
```

# Example

```
def find_product_list(input_list):
    total = 1                      # O(1) - create a counter
    for integer in input_list:       # O(n) - iterate through every item
        total = total * integer      # O(1) - find product, assign to total
    return total                     # O(1) - return total
```

- ❖ Time complexity is  $O(n)$  because it scales with the size of the input

# Example



- ❖ Time complexity is  $O(n)$  because here 2 is a constant

# Example

```
void printHi100Times(int arr[], int size)
{
    printf("First element of array = %d\n", arr[0]);

    for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

- ❖ Time complexity is  $O(1+n/2+100)$  but we can call it  $O(n)$
- ❖ We only look for the big O notation when n gets arbitrarily large. As n gets big, adding 100 or dividing by two decreases significantly

# Example

Cost Time require for line ( Units )	Repeation No. of Times Executed	Total Total Time required in worst case
1	1	1
$1 + 1 + 1$	$1 + (n+1) + n$	$2n + 2$
2	n	$2n$
1	1	1
		<b><math>4n + 4</math></b> Total Time required

- ❖ If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity of  $O(n)$

# Example

```
main ()  
{  
    x = y + z;           → O(1)  
    for (i = 1; i < n; i++)  
    {  
        a = b + c;       → O(n)  
    }  
}  
}
```

Big "O":  $O(1) + O(n)$   
 $= O(1 + n)$   
Thus by ignoring the constant  
 $O(1)$ , We can write:  
 $= O(n)$

# Example

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

int main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        cout << "Hello World !!!\n";
    }
    return 0;
}
```

## OUTPUT

```
Hello World !!!
```

- ❖ In the above code “Hello World !!!” is printed only n times on the screen, as the value of n can change
- ❖ The time complexity is linear: O(n) i.e. every time, a linear amount of time is required to execute code

# Time complexity - $O(n \log n)$

- It should be quite clear from the notation itself, it is a combination of Linear and Logarithmic Time Complexities
- The time taken by this is slightly less than the linear time complexity but not as slow as the quadratic time complexity
- Heap sort and quick sort are examples

# Time complexity - $O(n^2)$

- When the running time of an algorithm increases non-linearly  $O(n^2)$  with the length of the input, it is assumed to have non-linear time complexity
- In general, nested loops fall into the  $O(n)*O(n) = O(n^2)$  time complexity order, where one loop takes  $O(n)$  and if the function includes loops inside loops, it takes  $O(n)*O(n) = O(n^2)$

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++) cout<<"Hello dev!!";
    }
}
```

```
3
2
Hello dev!!
Hello dev!!
Hello dev!!
Hello dev!!
Hello dev!!
Hello dev!!
```

# Example

```
def find_product_matrix(input_matrix):
    total = 1                                     # O(1) - create a counter
    for input_list in input_matrix:                # O(n^2) - iterate through every list
        for integer in input_list:                  # iterate through every item in each list
            total = total * integer                 # O(1) - find product, assign to total
    return total                                    # O(1) - return total
```

- ❖ Time complexity is **O(n<sup>2</sup>)** because we need to iterate through n lists which each have a size of n

# Example

```
main ()  
{  
    x = y + z; O(1)  
      
    for ( i = 1; i <= n; i++ ) O(n)  
    {  
        for( j = 1 ; j <= n; j++ ) O(n)  
        {  
            a = b + c; O(1)  
        }  
    }  
}
```

- We can conclude that there is two Loop, Both, the Inner loop and the outer loop are dependent on the input size 'n'. So we have two  $O(n)$  and two constant  $O(1)$
- Here is how to calculate the time complexity:  
Big "O":  $( O(1) + O(n) + O(n) )$   
 $= O ( 1+ n + n )$
- Ignoring the constant  $O(1)$ , we get:  
 $= O ( n^2 )$
- Hence, the time complexity of the above program is  $O ( n^2 )$

# Example

```
const numbers = [1, 2, 3, 4, 5];

function logEverythingFiveTimes(items) {
  for (let i = 0; i < items.length; i++) { // O(n)
    for (let j = 0; j < items.length; j++) { // O(n)
      console.log(items[i]) // O(1)
    }
  }
}
```

- In this case, we multiply  $O(n) * O(n)$  instead of adding the run times together
- Ignoring the constant  $O(1)$ , we get:  
 $= O(n^2)$
- Hence, the time complexity of the above program is  $O(n^2)$

# Example

```
const numbers = [1, 2, 3, 4, 5];

function printMultiplesThenSum(items) {
  for (let i = 0; i < items.length; i++) { // O(n)
    for (let j = 0; j < items.length; j++) { // O(n)
      console.log(items[i]); // O(1)
    }
  }

  const sum = items.reduce((acc, item) => { // O(n)
    return acc += item; // O(1)
  }, 0);

  return sum; // O(1)
}
```

- we can see that the Big-O notation for this function would be  $O(n^2 + n)$ . Since Big-O is also not concerned with non-dominant terms, we drop the  $n$  (quadratic wins since it is worse than linear time)
- Throwing out non-dominant terms is the second rule to follow when analyzing the run time of an algorithm
- In the end,  $O(n^2 + n)$  gives us  $O(n^2)$

# Example

```
f ()  
{  
for (int i = 1; i <=n; i += 2)  
{  
for (int j = 1; j <=n; j += 3)  
{  
for (int k=1; k<=j; k++)  
{  
printf("DATA STRUCTURES");  
}  
}  
}  
}
```

$O(N^3)$

Code Analysis:

for (int i = 1; i <=n; i += 2)  $O(N)$  Level: 1

for (int j = 1; j <=n; j += 3)  $O(N)$  Level: 2

for (int k=1; k<=j; k++)  $O(N)$  Level: 3

## Example: Sum of N Numbers(Time Complexity)

Algorithm	Step Count
Alg sum(a[],n)	--
{	--
sum=0;	1
for(i=1 to n) do	N+1
sum=sum+a[i];	N
return sum;	1
}	
TOTAL	2N+3
	O(N)

# Example

```
function addUpTo(n) {  
    return n * (n + 1) / 2;  
}
```

Always 3 operations  
**O(1)**

```
function addUpTo(n) {  
    let total = 0;  
    for (let i = 1; i <= n; i++) {  
        total += i;  
    }  
    return total;  
}
```

Number of operations is (eventually)  
bounded by a multiple of  $n$  (say,  $10n$ )  
**O( $n$ )**

# Example

```
function countUpAndDown(n) {  
    console.log("Going up!");  
    for (let i = 0; i < n; i++) {  
        console.log(i);  
    }  
    console.log("At the top!\\nGoing down...");  
    for (let j = n - 1; j >= 0; j--) {  
        console.log(j);  
    }  
    console.log("Back down. Bye!");  
}
```

Number of operations is  
(eventually) bounded by a  
multiple of  $n$  (say,  $10n$ )

**$O(n)$**

```
function printAllPairs(n) {  
    for (var i = 0; i < n; i++) {  
        for (var j = 0; j < n; j++) {  
            console.log(i, j);  
        }  
    }  
}
```

$O(n)$  operation inside of an  
 $O(n)$  operation.

**$O(n^2)$**

# Example

```
#include <stdio.h>
int main()
{
    int a = 4;
    int b = 6;
    int c;
    c = a + b;
    printf("%d, c);
}
```

- **The program has an  $O(1)$  time complexity since it only contains assignment and arithmetic operations, which will all be run only once**

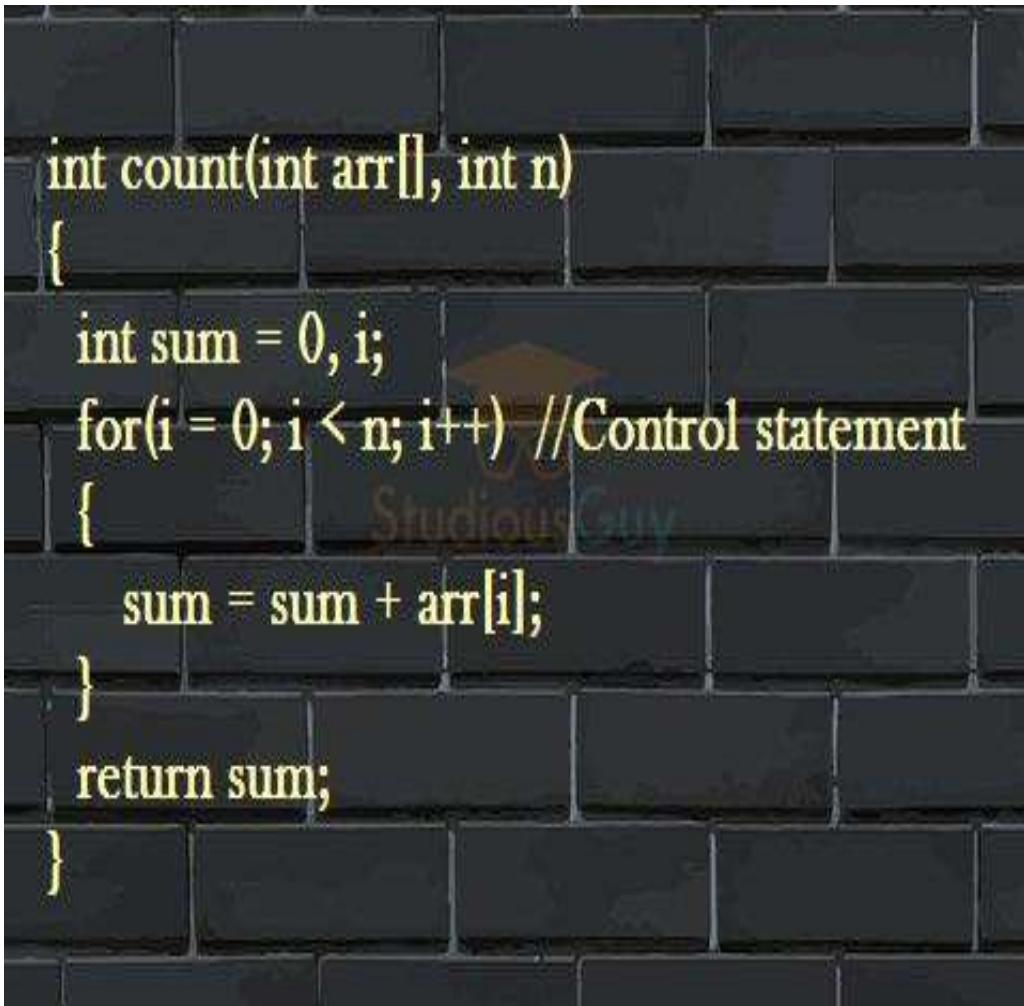
# Example



```
void printFirstElementOfArray(int arr[])
{
    printf("First element of array = %d",arr[0]);
}
```

- **Concerning its input, this function executes in O(1) time**
- **The input array could contain one item or 1,000, but this function only requires one step**

# Example



```
int count(int arr[], int n)
{
    int sum = 0, i;
    for(i = 0; i < n; i++) //Control statement
    {
        sum = sum + arr[i];
    }
    return sum;
}
```

- We have a control statement in the code that executes  $n$  times
- There are other operations like assignment, arithmetic, and a return statement.
- The time complexity is hence  $O(n + 3)$
- The constant values become insignificant as  $n$  is higher. As a result, if a program just has a control statement, the complexity of assignment, arithmetic, logic, and return statements might be neglected
- As a result, the above-mentioned code's final time complexity is  $O(n)$

# Example

```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
    }
}
```

- We're stacking two loops here. If we have  $n$  items in our array, our outer loop will run  $n$  times, and our inner loop will run  $n$  times for each iteration of the outer loop, totaling  $n^2$  prints. As a result, this function takes  $O(n^2)$  time to complete (or “quadratic time”)

# Factorial Complexity: O(n!)

The factorial of 5, or  $5!$  is:



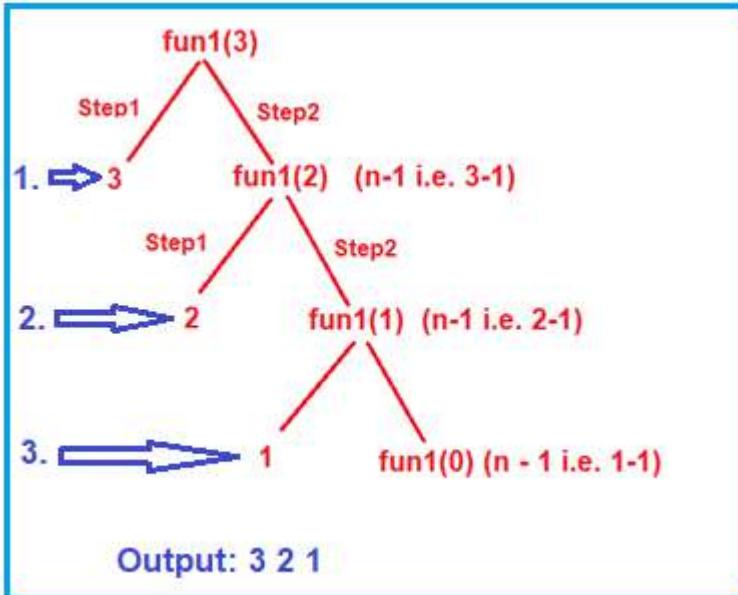
A blackboard with a chalkboard texture. In the center, the equation  $5 * 4 * 3 * 2 * 1 = 120$  is written in yellow chalk. The equals sign and the result are in white chalk.

$$5 * 4 * 3 * 2 * 1 = 120$$

- If Big O assists us in identifying the worst-case situation for our algorithms, then  $O(n!)$  is the worst of the worst

# Example

```
void fun1(int n)
{
    if(n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}
void main()
{
    int x=3;
    fun1(x);
}
```



- The function (`fun1`) is just printing the value of  $n$  and it takes one unit of time for printing i.e.  $O(1)$
- Only one-time `printf` is written there. But this is a recursive function. So, it is calling itself again and again
- As you can see in the above tracing tree, first it prints the value 3, then print 2 and then print the value 1. That means the `printf` statement executed three times. So, this recursive function will take 3 units of time to execute when the  $n$  value is 3. If we make the  $n$  value is 5 then it will take 5 units of time to execute this recursive function
- The time can be represented as the order of  $n$  i.e.  $O(n)$ . The time taken is in order of  $n$

## Example: Sum of N Numbers(Space Complexity)

### Algorithm

Alg sum(a[],n)

A[]– n words

{

n=one word

sum=0;

Sum=one word

for(i=1 to n) do

i= one word

sum=sum+a[i];

return sum;

}

TOTAL

N+3

O(n)

# Try Some Other examples

- Matrix Addition
- Matrix Multiplication
- Bubble sort
- Insertion Sort
- Selection Sort
- Linear Search
- Binary search

## Chapter 3

# Time complexity

Use of time complexity makes it easy to estimate the running time of a program. Performing an accurate calculation of a program's operation time is a very labour-intensive process (it depends on the compiler and the type of computer or speed of the processor). Therefore, we will not make an accurate measurement; just a measurement of a certain order of magnitude.

Complexity can be viewed as the maximum number of primitive operations that a program may execute. Regular operations are single additions, multiplications, assignments etc. We may leave some operations uncounted and concentrate on those that are performed the largest number of times. Such operations are referred to as *dominant*.

The number of dominant operations depends on the specific input data. We usually want to know how the performance time depends on a particular aspect of the data. This is most frequently the data size, but it can also be the size of a square matrix or the value of some input variable.

### 3.1: Which is the dominant operation?

```
1 def dominant(n):
2     result = 0
3     for i in xrange(n):
4         result += 1
5     return result
```

---

The operation in line 4 is dominant and will be executed  $n$  times. The complexity is described in Big-O notation: in this case  $O(n)$  — *linear* complexity.

The complexity specifies the order of magnitude within which the program will perform its operations. More precisely, in the case of  $O(n)$ , the program may perform  $c \cdot n$  operations, where  $c$  is a constant; however, it may not perform  $n^2$  operations, since this involves a different order of magnitude of data. In other words, when calculating the complexity we omit constants: i.e. regardless of whether the loop is executed  $20 \cdot n$  times or  $\frac{n}{5}$  times, we still have a complexity of  $O(n)$ , even though the running time of the program may vary. When analyzing the complexity we must look for specific, worst-case examples of data that the program will take a long time to process.

### 3.1. Comparison of different time complexities

Let's compare some basic time complexities.

#### 3.2: Constant time — $O(1)$ .

```
1 def constant(n):
2     result = n * n
3     return result
```

---

There is always a fixed number of operations.

#### 3.3: Logarithmic time — $O(\log n)$ .

```
1 def logarithmic(n):
2     result = 0
3     while n > 1:
4         n /= 2
5         result += 1
6     return result
```

---

The value of  $n$  is halved on each iteration of the loop. If  $n = 2^x$  then  $\log n = x$ . How long would the program below take to execute, depending on the input data?

#### 3.4: Linear time — $O(n)$ .

```
1 def linear(n, A):
2     for i in xrange(n):
3         if A[i] == 0:
4             return 0
5     return 1
```

---

Let's note that if the first value of array  $A$  is 0 then the program will end immediately. But remember, when analyzing time complexity we should check for worst cases. The program will take the longest time to execute if array  $A$  does not contain any 0.

#### 3.5: Quadratic time — $O(n^2)$ .

```
1 def quadratic(n):
2     result = 0
3     for i in xrange(n):
4         for j in xrange(i, n):
5             result += 1
6     return result
```

---

The result of the function equals  $\frac{1}{2} \cdot (n \cdot (n + 1)) = \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n$  (the explanation is in the exercises). When calculating the complexity we are interested in a term that grows fastest, so we not only omit constants, but also other terms ( $\frac{1}{2} \cdot n$  in this case). Thus we get quadratic time complexity. Sometimes the complexity depends on more variables (see example below).

#### 3.6: Linear time — $O(n + m)$ .

```
1 def linear2(n, m):
2     result = 0
3     for i in xrange(n):
4         result += i
5     for j in xrange(m):
6         result += j
7     return result
```

## Exponential and factorial time

It is worth knowing that there are other types of time complexity such as factorial time  $O(n!)$  and exponential time  $O(2^n)$ . Algorithms with such complexities can solve problems only for very small values of  $n$ , because they would take too long to execute for large values of  $n$ .

## 3.2. Time limit

Nowadays, an average computer can perform  $10^8$  operations in less than a second. Sometimes we have the information we need about the expected time complexity (for example, Codility specifies the expected time complexity), but sometimes we do not.

The time limit set for online tests is usually from 1 to 10 seconds. We can therefore estimate the expected complexity. During contests, we are often given a limit on the size of data, and therefore we can guess the time complexity within which the task should be solved. This is usually a great convenience because we can look for a solution that works in a specific complexity instead of worrying about a faster solution. For example, if:

- $n \leq 1\,000\,000$ , the expected time complexity is  $O(n)$  or  $O(n \log n)$ ,
- $n \leq 10\,000$ , the expected time complexity is  $O(n^2)$ ,
- $n \leq 500$ , the expected time complexity is  $O(n^3)$ .

Of course, these limits are not precise. They are just approximations, and will vary depending on the specific task.

## 3.3. Space complexity

Memory limits provide information about the expected space complexity. You can estimate the number of variables that you can declare in your programs.

In short, if you have constant numbers of variables, you also have constant space complexity: in Big-O notation this is  $O(1)$ . If you need to declare an array with  $n$  elements, you have linear space complexity —  $O(n)$ .

More specifically, space complexity is the amount of memory needed to perform the computation. It includes all the variables, both global and local, dynamic pointer data-structures and, in the case of recursion, the contents of the stack. Depending on the convention, input data may also be included. Space complexity is more tricky to calculate than time complexity because not all of these variables and data-structures may be needed at the same time. Global variables exist and occupy memory all the time; local variables (and additional information kept on the stack) will exist only during invocation of the function.

## 3.4. Exercise

**Problem:** You are given an integer  $n$ . Count the total of  $1 + 2 + \dots + n$ .

**Solution:** The task can be solved in several ways. Some person, who knows nothing about time complexity, may implement an algorithm in which the result is incremented by 1:

### 3.7: Slow solution — time complexity $O(n^2)$ .

```
1 def slow_solution(n):
2     result = 0
3     for i in xrange(n):
4         for j in xrange(i + 1):
5             result += 1
6     return result
```

---

Another person may increment the result respectively by  $1, 2, \dots, n$ . This algorithm is much faster:

### 3.8: Fast solution — time complexity $O(n)$ .

```
1 def fast_solution(n):
2     result = 0
3     for i in xrange(n):
4         result += (i + 1)
5     return result
```

---

But the third person's solution is even quicker. Let us write the sequence  $1, 2, \dots, n$  and repeat the same sequence underneath it, but in reverse order. Then just add the numbers from the same columns:

1	2	3	...	$n - 1$	$n$
$n$	$n - 1$	$n - 2$	...	2	1
$n + 1$	$n + 1$	$n + 1$	...	$n + 1$	$n + 1$

The result in each column is  $n + 1$ , so we can easily count the final result:

### 3.9: Model solution — time complexity $O(1)$ .

```
1 def model_solution(n):
2     result = n * (n + 1) // 2
3     return result
```

---

# Data Structures and Algorithms (BCSE202L)

*Module 1: Algorithm Analysis*

Dr. Priyanka N

# Asymptotic Notations

- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value
- There are mainly three asymptotic notations:
  - Big-O notation ( $O$  Notation)
  - Omega notation ( $\Omega$  Notation)
  - Theta notation ( $\Theta$  Notation)

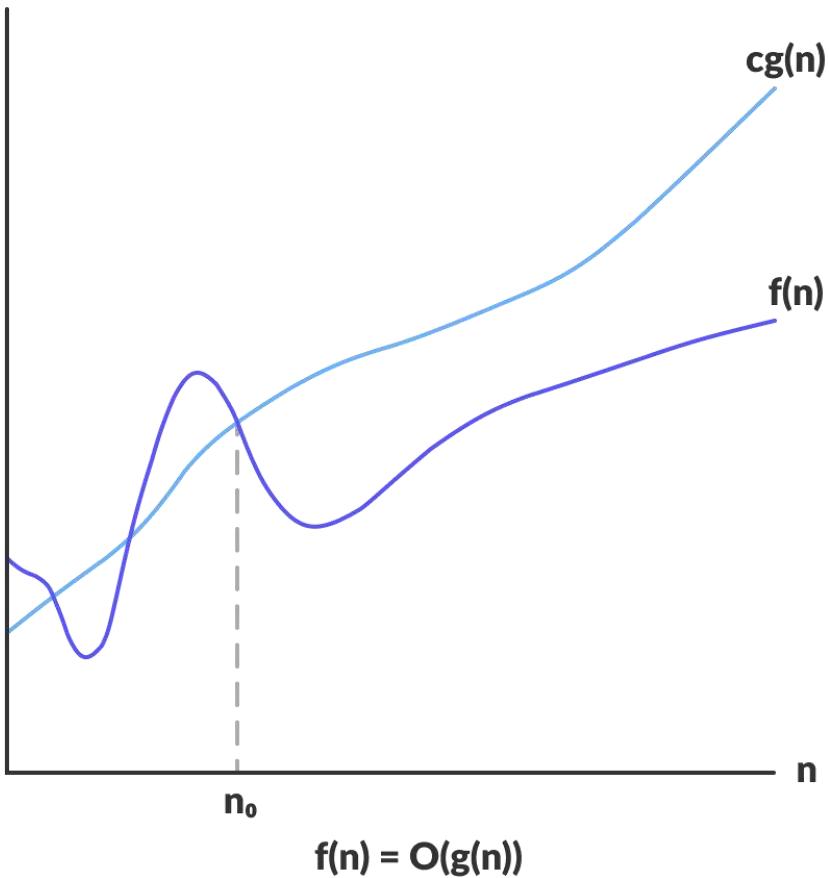
# Big-O Notation (O-notation)

- Big-O notation represents the upper bound of the running time of an algorithm
- It gives the worst-case complexity or the longest amount of time an algorithm can possibly take to complete
- **Mathematical Representation of Big-O Notation:**

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

- The above expression can be described as a function  $f(n)$  belonging to the set  $O(g(n))$  if there exists a positive constant  $c$  such that it lies between  $0$  and  $c.g(n)$ , for sufficiently large  $n$

# Big-O Notation (O-notation)



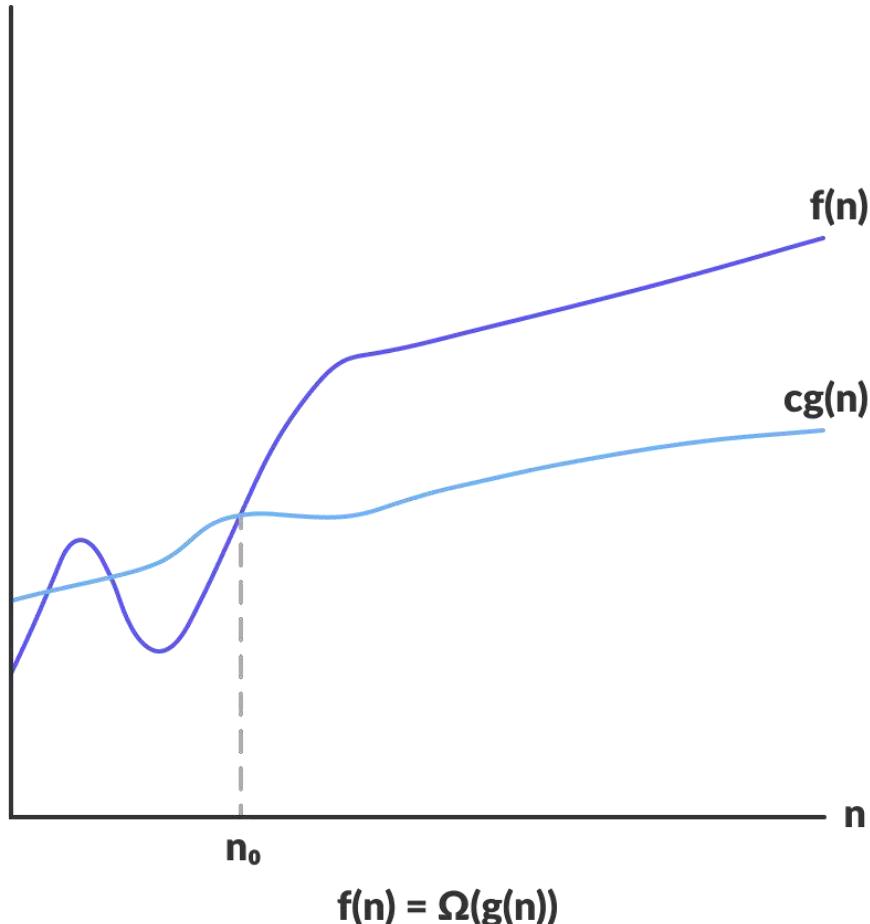
# Omega notation ( $\Omega$ Notation)

- Omega notation represents the lower bound of the running time of an algorithm
- It provides the best-case complexity of an algorithm
- **Mathematical Representation of Omega Notation:**

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

- The above expression can be described as a function  $f(n)$  belonging to the set  $\Omega(g(n))$  if there exists a positive constant  $c$  such that it lies above  $c.g(n)$ , for sufficiently large  $n$
- For any value of  $n$ , the minimum time required by the algorithm is given by Omega  $\Omega(g(n))$

# Omega notation ( $\Omega$ Notation)



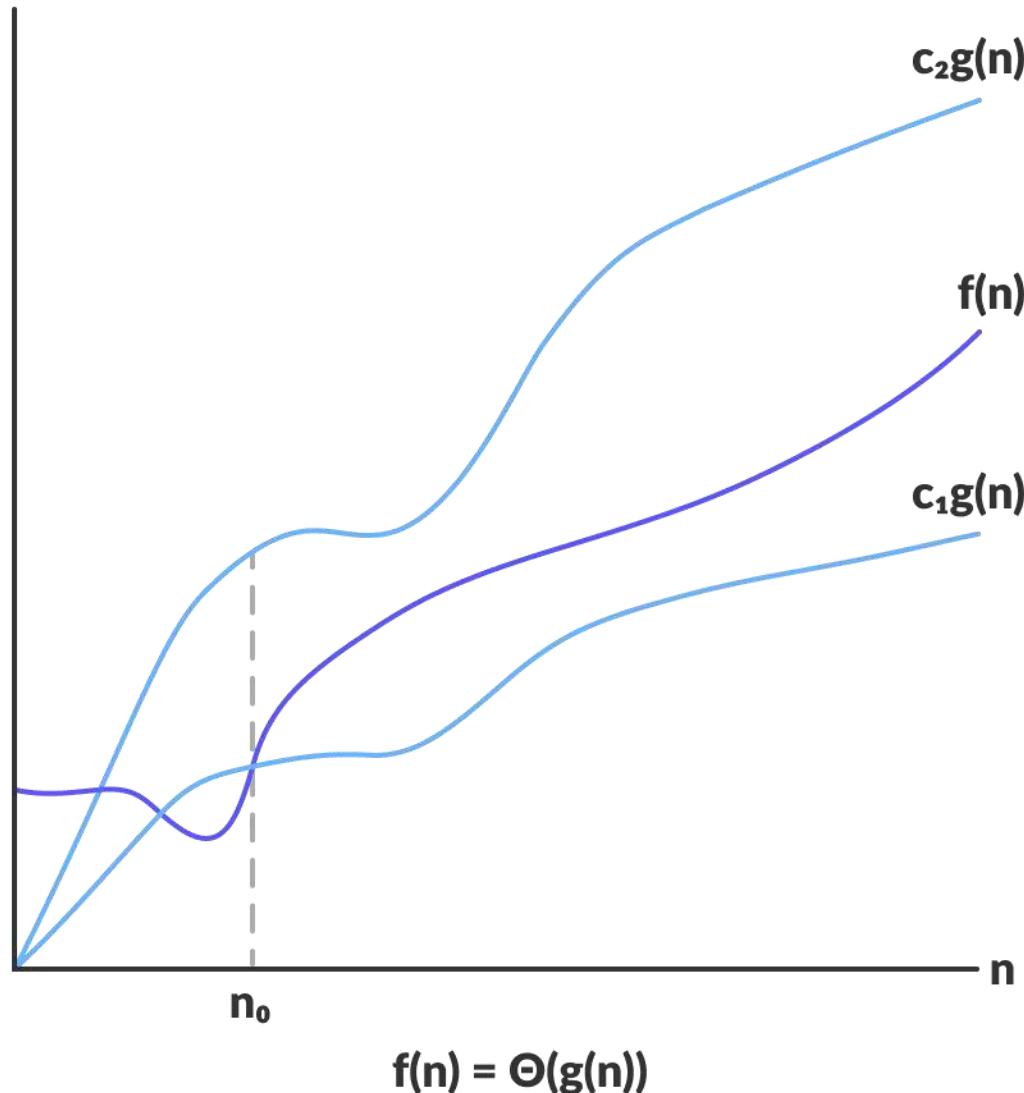
# Theta Notation ( $\Theta$ -notation)

- Theta notation encloses the function from above and below
- Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm
- **Mathematical Representation of Omega Notation:**

$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n) \text{ for all } n \geq n_0 \}$

- The above expression can be described as a function  $f(n)$  belongs to the set  $\Theta(g(n))$  if there exist positive constants  $c_1$  and  $c_2$  such that it can be sandwiched between  $c_1.g(n)$  and  $c_2.g(n)$ , for sufficiently large  $n$
- If a function  $f(n)$  lies anywhere in between  $c_1.g(n)$  and  $c_2.g(n)$  for all  $n \geq n_0$ , then  $f(n)$  is said to be asymptotically tight bound

# Theta Notation ( $\Theta$ -notation)



# Worst-case, Average-case, Best-case, and Amortized Time Complexity

- Generally, we perform the following types of analysis:
  - ❑ **Worst-case** – The maximum number of steps taken on any instance of size  $a$
  - ❑ **Best-case** – The minimum number of steps taken on any instance of size  $a$
  - ❑ **Average case** – An average number of steps taken on any instance of size  $a$

# Worst-case Time Complexity

- This denotes the behavior of an algorithm with respect to the worst possible case of the input instance
- The worst-case running time of an algorithm is an upper bound on the running time for any input
- Having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit

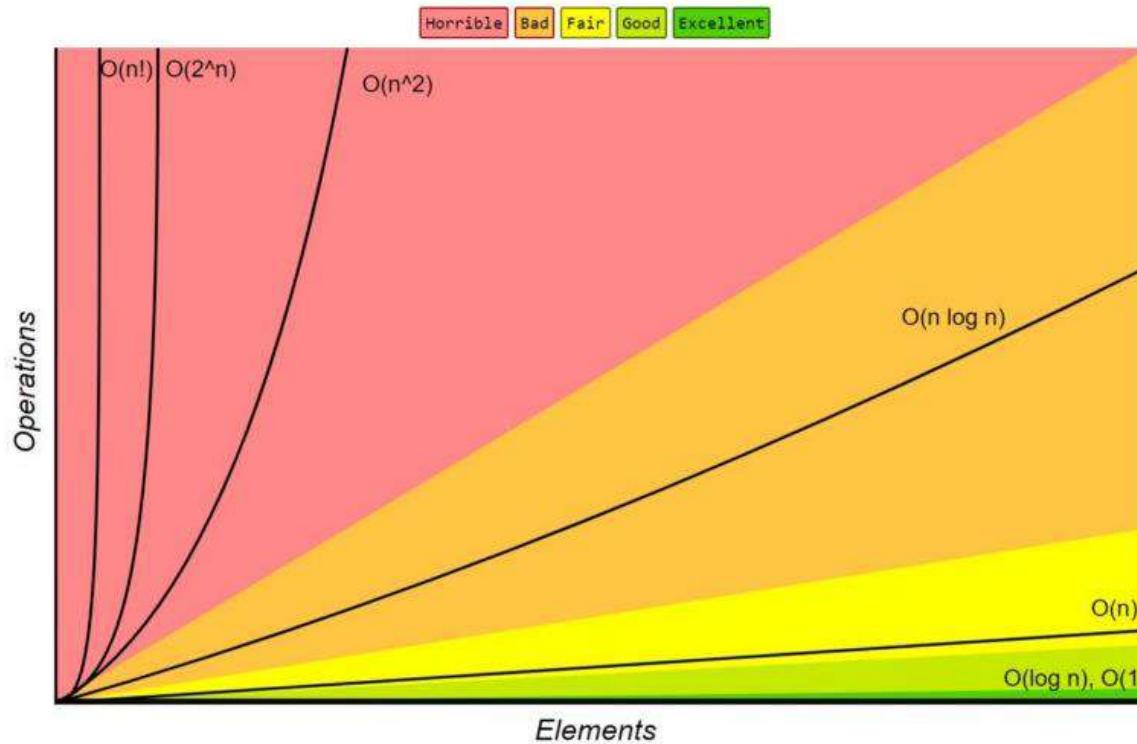
# Average-case Time Complexity

- The average-case running time of an algorithm is an estimate of the running time for an ‘average’ input
- It specifies the expected behavior of the algorithm when the input is randomly drawn from a given distribution
- Average-case running time assumes that all inputs of a given size are equally likely

# Best-case Time Complexity

- The term ‘best-case performance’ is used to analyze an algorithm under optimal conditions
- For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list
- However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance
- It is always recommended to improve the average performance and the worst-case performance of an algorithm

# Order of Growth



- **Order of Growth:**

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n < n^n$$

# UPPERBOUND

$$1. \quad f(n) = 6n + 3$$

## Solution

To find upper bound of  $f(n)$ , we have to find  $c$  and  $n_0$  such that  $0 \leq f(n) \leq c \times g(n)$  for all  $n \geq n_0$

$$0 \leq f(n) \leq c \times g(n)$$

$$0 \leq 6n + 3 \leq c \times g(n)$$

$$0 \leq 6n + 3 \leq 6n + 3n, \text{ for all } n \geq 1 \text{ (There can be such infinite possibilities)}$$

$$0 \leq 6n + 3 \leq 9n$$

$$\text{So, } c = 9 \text{ and } g(n) = n, n_0 = 1$$

## Tabular Approach

$$0 \leq 6n + 3 \leq c \times g(n)$$

$$0 \leq 6n + 3 \leq 7n$$

Now, manually find out the proper  $n_0$ , such that  $f(n) \leq c \cdot g(n)$

n	$f(n) = 6n + 3$	$c \cdot g(n) = 7n$
1	9	7
2	15	14
3	21	21
4	27	28
5	33	35

From Table, for  $n \geq 3$ ,  $f(n) \leq c \times g(n)$  holds true. So,  $c = 7$ ,  $g(n) = n$  and  $n_0 = 3$ , There can be such multiple pair of  $(c, n_0)$

$$f(n) = O(g(n)) = O(n) \text{ for } c = 9, n_0 = 1$$

$$f(n) = O(g(n)) = O(n) \text{ for } c = 7, n_0 = 3$$

$$2. \quad f(n) = 3n^2 + 2n + 4$$

### Solution:

To find upper bound of  $f(n)$ , we have to find  $c$  and  $n_0$  such that  $0 \leq f(n) \leq c \times g(n)$  for all  $n \geq n_0$

$$0 \leq f(n) \leq c \times g(n)$$

$$0 \leq 3n^2 + 2n + 4 \leq c \times g(n)$$

$$0 \leq 3n^2 + 2n + 4 \leq 3n^2 + 2n^2 + 4n^2,$$

for all  $n \geq 1$ :

$$0 \leq 3n^2 + 2n + 4 \leq 9n^2$$

$$0 \leq 3n^2 + 2n + 4 \leq 9n^2$$

So,  $c = 9$ ,  $g(n) = n^2$  and  $n_0 = 1$

### Tabular Approach

$$0 \leq 3n^2 + 2n + 4 \leq c.g(n)$$

$$0 \leq 3n^2 + 2n + 4 \leq 4n^2$$

Now, manually find out the proper  $n_0$ , such that  $f(n) \leq c.g(n)$

<b>n</b>	<b><math>f(n) = 3n^2 + 2n + 4</math></b>	<b><math>c.g(n) = 4n^2</math></b>
1	9	4
2	20	16
3	37	36
4	60	64
5	89	100

From Table, for  $n \geq 4$ ,  $f(n) \leq c \times g(n)$  holds true. So,  $c = 4$ ,  $g(n) = n^2$  and  $n_0 = 4$ . There can be such multiple pair of  $(c, n_0)$

$$f(n) = O(g(n)) = O(n^2) \text{ for } c = 9, n_0 = 1$$

$$f(n) = O(g(n)) = O(n^2) \text{ for } c = 4, n_0 = 4$$

$$3. \quad f(n) = 2n^3 + 4n + 5$$

## Solution:

To find upper bound of  $f(n)$ , we have to find  $c$  and  $n_0$  such that  $0 \leq f(n) \leq c \times g(n)$  for all  $n \geq n_0$

$$0 \leq f(n) \leq c \cdot g(n)$$

$$0 \leq 2n^3 + 4n + 5 \leq c \times g(n)$$

$$0 \leq 2n^3 + 4n + 5 \leq 2n^3 + 4n^3 + 5n^3, \text{ for all } n \geq 1$$

$$0 \leq 2n^3 + 4n + 5 \leq 11n^3$$

$$\text{So, } c = 11, g(n) = n^3 \text{ and } n_0 = 1$$

## Tabular Approach

$$0 \leq 2n^3 + 4n + 5 \leq c \times g(n)$$

$$0 \leq 2n^3 + 4n + 5 \leq 3n^3$$

Now, manually find out the proper  $n_0$ , such that  $f(n) \leq c \times g(n)$

$n$	$f(n) = 2n^3 + 4n + 5$	$c \cdot g(n) = 3n^3$
1	11	3
2	29	24
3	71	81
4	149	192

From Table, for  $n \geq 3$ ,  $f(n) \leq c \times g(n)$  holds true. So,  $c = 3$ ,  $g(n) = n^3$  and  $n_0 = 3$ . There can be such multiple pair of  $(c, n_0)$

$$f(n) = O(g(n)) = O(n^3) \text{ for } c = 11, n_0 = 1$$

$$f(n) = O(g(n)) = O(n^3) \text{ for } c = 3, n_0 = 3 \text{ and so on.}$$

# LOWERBOUND

$$1. \quad f(n) = 6n + 3$$

## Solution:

To find lower bound of  $f(n)$ , we have to find  $c$  and  $n_0$  such that  $0 \leq c \cdot g(n) \leq f(n)$  for all  $n \geq n_0$

$$0 \leq c \cdot g(n) \leq f(n)$$

$$0 \leq c \cdot g(n) \leq 6n + 3$$

$$0 \leq 6n \leq 6n + 3 \rightarrow \text{true, for all } n \geq n_0$$

$$0 \leq 5n \leq 6n + 3 \rightarrow \text{true, for all } n \geq n_0$$

Above both inequalities are true and there exists such infinite inequalities. So,

$$f(n) = \Omega(g(n)) = \Omega(n) \text{ for } c = 6, n_0 = 1$$

$$f(n) = \Omega(g(n)) = \Omega(n) \text{ for } c = 5, n_0 = 1$$

$$2. \quad f(n) = 2n^3 + 4n + 5$$

## Solution:

To find lower bound of  $f(n)$ , we have to find  $c$  and  $n_0$  such that  $0 \leq c \cdot g(n) \leq f(n)$  for all  $n \geq n_0$

$$0 \leq c \cdot g(n) \leq f(n)$$

$$0 \leq c \cdot g(n) \leq 2n^3 + 4n + 5$$

$$0 \leq 2n^3 \leq 2n^3 + 4n + 5 \rightarrow \text{true, for all } n \geq 1$$

$$0 \leq n^3 \leq 2n^3 + 4n + 5 \rightarrow \text{true, for all } n \geq 1$$

Above both inequalities are true and there exists such infinite inequalities.

So,  $f(n) = \Omega(g(n)) = \Omega(n^3)$  for  $c = 2, n_0 = 1$

$f(n) = \Omega(g(n)) = \Omega(n^3)$  for  $c = 1, n_0 = 1$

## TIGHTBOUND

1.  $f(n) = 6n + 3$

### Solution:

To find tight bound of  $f(n)$ , we have to find  $c_1, c_2$  and  $n_0$  such that,  $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$  for all  $n \geq n_0$

$$0 \leq c_1 \cdot g(n) \leq 6n + 3 \leq c_2 \cdot g(n)$$

$$0 \leq 5n \leq 6n + 3 \leq 9n, \text{ for all } n \geq 1$$

Above inequality is true and there exists such infinite inequalities.

So,  $f(n) = \Theta(g(n)) = \Theta(n)$  for  $c_1 = 5, c_2 = 9, n_0 = 1$

$$2. \quad f(n) = 3n^2 + 2n + 4$$

### Solution:

To find tight bound of  $f(n)$ , we have to find  $c_1$ ,  $c_2$  and  $n_0$  such that,  $0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$  for all  $n \geq n_0$

$$0 \leq c_1 \times g(n) \leq 3n^2 + 2n + 4 \leq c_2 \times g(n)$$

$$0 \leq 3n^2 \leq 3n^2 + 2n + 4 \leq 9n^2, \text{ for all } n \geq 1$$

Above inequality is true and there exists such infinite inequalities. So,

$$f(n) = \Theta(g(n)) = \Theta(n^2) \text{ for } c_1 = 3, c_2 = 9, n_0 = 1$$

$$3. \quad f(n) = 2n^3 + 4n + 5$$

### Solution:

To find tight bound of  $f(n)$ , we have to find  $c_1$ ,  $c_2$  and  $n_0$  such that,  $0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$  for all  $n \geq n_0$

$$0 \leq c_1 \times g(n) \leq 2n^3 + 4n + 5 \leq c_2 \times g(n)$$

$$0 \leq 2n^3 \leq 2n^3 + 4n + 5 \leq 11n^3, \text{ for all } n \geq 1$$

Above inequality is true and there exists such infinite inequalities. So,

$$f(n) = \Theta(g(n)) = \Theta(n^3) \text{ for } c_1 = 2, c_2 = 11, n_0 = 1$$

## ①

# MASTER THEOREM FOR DIVIDING FUNCTIONS

- 1) The master theorem is used in calculating the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way.
- 2) The master method is a formula for solving recurrence relations of the form:

$$T(n) = aT(n/b) + f(n)$$

where,

$n$  = size of input

$a$  = number of subproblems in the recursion

$n/b$  = size of each subproblem. All subproblems are assumed to have the same size.

$f(n)$  = cost of the work done outside the recursive call, which includes the cost of dividing the problem & cost of merging the solutions.

- 3) Here,  $a \geq 1$  and  $b > 1$  are constants, and  $f(n)$  is an asymptotically positive function

(2)

$$f(n) = O(n^k \log^P n)$$

→ first, we have to find two values:

$$\textcircled{1} \quad \log_b^a$$

$$\textcircled{2} \quad k$$

→ There are three cases:

Case 1:

$$\text{if } \log_b^a > k \text{ then } O(n^{\log_b^a})$$

Case 2:

$$\text{if } \log_b^a = k$$

$$(i) \text{ if } P > -1 \text{ then } O(n^k \log^{P+1} n)$$

$$(ii) \text{ if } P = -1 \text{ then } O(n^k \log \log n)$$

$$(iii) \text{ if } P < -1 \text{ then } O(n^k)$$

Case 3:

$$\text{if } \log_b^a < k$$

$$(i) \text{ if } P \geq 0 \text{ then } O(n^k \log^P n)$$

$$(ii) \text{ if } P < 0 \text{ then } O(n^k)$$

Examples (Case 1) : Solve the following recurrence relation using master's theorem.

$$\textcircled{1} \quad T(n) = 2T\left(\frac{n}{2}\right) + 1$$

Sol:

Here, 
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

(i)  $a = 2, b = 2$

$$f(n) = \Theta(1)$$

$$= \Theta(n^0 \log^0 n)$$

$$\therefore f(n) = O(n^k, \log^p n)$$

Here,  $k = 0, p = 0$

(ii)  $\log_b^a = \log_2^2$

$$= 1$$

$$\log_b^a > k, \quad | > 0$$

It satisfies case 1.

Hence,

$$T(n) = O(n^1)$$

$$\textcircled{2} \quad T(n) = 4T(n/2) + n$$

Sol:

$$\boxed{T(n) = aT(n/b) + f(n)}$$
$$f(n) = O(n^k \log^p n)$$

Here,

$$(i) \quad a = 4, b = 2$$

$$f(n) = n^1 \log^0 n$$

$$K=1, P=0$$

$$(ii) \quad \log_b a = \log_2 4 = 2$$

$$2 > 1$$

Hence, case 1 satisfied.

Hence,

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

$$\textcircled{3} \quad T(n) = 8T(n/2) + n$$

Sol:

$$(i) \quad a = 8, b = 2$$

$$f(n) = n^1 \log^0 n$$

$$K = 1, P = 0$$

$$(ii) \log_b a = \log_2 8 = 3$$

$\boxed{3 > 1}$ . Hence case 1 is satisfied.

$$T(n) = \Theta(n^3)$$

$$\textcircled{4} \quad T(n) = 8T(n/2) + n^2$$

Sol:

$$(i) \quad a = 8, b = 2$$

$$f(n) = n^2 \log^0 n$$

$$K = 2, P = 0$$

$$(ii) \log_b a = \log_2 8 = 3$$

$\boxed{3 > 2}$ . Hence case 1 is satisfied.

$$T(n) = \Theta(n^3)$$

## Examples: case 2

$$\textcircled{1} \quad T(n) = 2T(n/2) + n$$

Sol:

$$(i) \quad a=2, b=2$$

$$f(n) = n^k \log^p n$$

$$k=1, p=0$$

$$(ii) \quad \log_b a = \log_2 2 = 1$$

$$\begin{cases} \log_b a = k \\ l = 1 \end{cases}$$

Hence, Case 2 is satisfied.

$$(iii) \quad p > -1$$

$$0 > -1$$

$$\text{Hence, } \Theta(n^k \log^{p+1} n)$$

$$\text{i.e. } \Theta(n^k \log n)$$

$$T(n) = \Theta(n \log n)$$

$$\textcircled{2} \quad T(n) = 2T(n/2) + n \log n \quad \textcircled{7}$$

Sol:

(i)  $a = 2, b = 2$

$$K = 1, P = 1$$

(ii)  $\log_b a = \log_2 2 = 1$

$$\begin{aligned} \log_b a &= K \\ 1 &= 1 \end{aligned} \quad \left. \begin{array}{l} \text{Hence, base 2 is} \\ \text{satisfied} \end{array} \right\}$$

(iii) ~~P need~~  $P > -1$

$$1 > -1$$

Hence  $T(n) = \Theta(n \log^2 n)$

$$\textcircled{3} \quad T(n) = 4T(n/2) + n^2$$

Sol:

(i)  $a = 4, b = 2, K = 2, P = 0$

(ii)  $\log_b a = \log_2 4 = 2$

$$\log_b a = K$$

$$2 = 2$$

(iii)  $P > -1$

$$0 > -1$$

Hence  $T(n) = \Theta(n^2 \log n)$

$$T(n) = 4T(n/2) + n^2 \log n$$

$$\Theta(n^2 \log^2 n)$$

$$\textcircled{4} \quad T(n) = 2T(n/2) + \frac{n}{\log n}$$

Sol:

(i)  $a=2, b=2, k=1, P=-1$  ( $n' \log^{-1} n$ )

(ii)  $\log_b^a = \log_2^2 = 1$

$$\begin{aligned} \log_b^a &= k \\ 1 &= 1 \end{aligned} \quad \left. \begin{array}{l} \text{Hence, case 2 is satisfied} \\ \text{Hence, } T(n) = \Theta(n' \log \log n) \end{array} \right.$$

(iii)  $P = -1$

$-1 = -1$

Hence,  $T(n) = \Theta(n' \log \log n)$

$$\textcircled{5} \quad T(n) = 2T(n/2) + \frac{n}{\log^2 n}$$

Sol:

(i)  $a=2, b=2, k=1, P=-2$

(ii)  $\log_b^a = \log_2^2 = 1$

$$\left. \begin{array}{l} \log_b a = k \\ 1 = 1 \end{array} \right\} \text{Hence, case 2 is satisfied.}$$

(9)

(iii)  $P < -1$

$$-2 < -1$$

Hence,  $\Theta(n)$

Examples: Case 3

$$\textcircled{1} \quad T(n) = T(n/2) + n^2$$

Sol:

$$(i) a = 1, b = 2, k = 2, P = 0$$

$$(ii) \log_b a = \log_2 1 = 0$$

$$\left. \begin{array}{l} \log_b a < k \\ 0 < 2 \end{array} \right\} \text{case 3 is satisfied}$$

(iii)  $P \geq 0$

$$\Theta(n^2 \log^0 n)$$

$$\therefore \Theta(n^2)$$

$$\textcircled{2} \quad T(n) = 2T(n/2) + n^2 \log^2 n$$

Sol:

(i)  $a=2, b=2, k=2, P=2$

(ii)  $\log_b a = \log_2 2 = 1$

$$\begin{cases} \log_b a \leq k \\ 1 < 2 \end{cases} \quad \left. \begin{array}{l} \text{Case 3 - n satisfied} \\ n \in (c_1 n^k, c_2 n^k] \end{array} \right.$$

(iii)  $P \geq 0$

$2 \geq 0$

Hence,  $\Theta(n^k \log^P n)$

$$T(n) = \Theta(n^2 \log^2 n)$$

# Algorithm Analysis

Module 1

**Dr. Priyanka N**

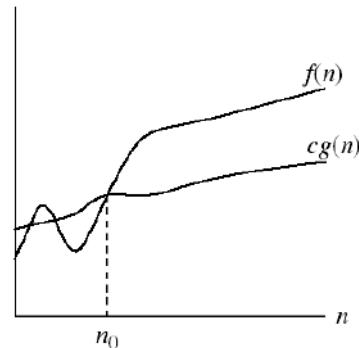
- What is  $f(n)$ 
  - time taken by the algorithm to execute this is not the real time it is an approximate time
- How to find it out  $f(n)$

## Two Types of algorithm

- Iterative
  - No of time loop executes
- Recursion
  - Backward Substitution

NOTE: CONSTANT O(1)

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



$g(n)$  is an *asymptotic lower bound* for  $f(n)$ .

# Algorithms for computing the Factorial

Recursion

```
int factorial (int n)
{
    If (n <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

Iterative

```
int factorial (int n)
{
    fact =1;
    if (n<=1)
        return 1;
    else {
        for (k=2; k<=n; k++)
            fact *= k;
        return fact;
    }
}
```

```
Function()
{
    int I;
    for(i=1 to n)
        print("Hello");
}
```

$O(n)$

```
Function()
{
    int I;
    for(i=1 to n)
        for(j=1 to n)
            print("Hello");
}
```

$O(n^2)$

```
function(){
    int i=1, s=1;
    while(S<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S
i
```

```
function(){
    int i=1, s=1;
    while(S<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S
i      1
```

```
function(){
    int i=1, s=1;
    while(S<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S      1
i      1
```

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
```

S	1	
i	1	2

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
```

S	1	3
i	1	2

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
```

S	1	3	
i	1	2	3

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
```

S	1	3	6
i	1	2	3

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S   1      3      6
i   1      2      3      4
```

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S  1      3      6      10
i  1      2      3      4
```

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
```

S	1	3	6	10	
i	1	2	3	4	5

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S  1      3      6      10     15
i  1      2      3      4      5
```

```
function(){
    int i=1, s=1;
    while(S<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S  1      3      6      10     15
i  1      2      3      4      5      6
```

```
function(){
    int i=1, s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S  1      3      6      10     15     21
i  1      2      3      4      5      6
```

```
function(){
    int i=1, s=1;
    while(S<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S   1      3      6      10     15     21
i   1      2      3      4      5      6.....n
```

```
function(){
    int i=1, s=1;
    while(S<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S   1     3     6     10    15    21.....n(n+1)/2 n>s stop
i   1     2     3     4     5     6.....n
```

```

function(){
    int i=1, s=1;
    while(S<=n)
    {
        i++;
        s=s+i;
        printf("hello")
    }
}
S      1      3      6      10     15      21.....k(k+1)/2
i      1      2      3      4      5      6.....k
K(K+1)/2 >=n
(K2+k)/2>=n

```

Which means  $K^2 = n$

$$k = \sqrt{n} \quad \text{ie } O(\sqrt{n})$$

```
Function(){  
    i=1;  
    for(i=1;i2<=n;i++)  
        printf("hello");  
}
```

$$i^2 \leq n$$

or  $i = \sqrt{n}$

$O(\sqrt{n})$ , or theta  $\sqrt{n}$  both are same

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
```

I

J

K

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
```

I        1  
J  
K

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}

I      1
J      1
K
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1
J      1
K      1*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2
J      1
K      1*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2
J      1      2
K      1*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2
J      1      2
K      1*100  2*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2      3
J      1      2
K      1*100  2*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2      3
J      1      2      3
K      1*100  2*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2      3
J      1      2      3
K      1*100  2*100  3*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2      3      4
J      1      2      3
K      1*100  2*100  3*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2      3      4
J      1      2      3      4
K      1*100  2*100  3*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
```

I        1        2        3        4

J        1        2        3        4

K        1\*100    2\*100    3\*100    4\*100

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
I      1      2      3      4      5
J      1      2      3      4
K      1*100  2*100  3*100  4*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}

I      1      2      3      4      5
J      1      2      3      4      5
K      1*100  2*100  3*100  4*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}

I      1      2      3      4      5
J      1      2      3      4      5
K      1*100  2*100  3*100  4*100  5*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
```

I	1	2	3	4	5.....n
J	1	2	3	4	5
K	1*100	2*100	3*100	4*100	5*100

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}

I      1      2      3      4      5.....n
J      1      2      3      4      5.....n
K      1*100  2*100  3*100  4*100  5*100
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}
```

I	1	2	3	4	5.....n
J	1	2	3	4	5.....n
K	1*100	2*100	3*100	4*100	5*100.....n*100

```

Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}

I      1      2      3      4      5.....n
J      1      2      3      4      5.....n
K      1*100  2*100  3*100  4*100  5*100.....n*100

```

$$1*100 + 2*100 + 3*100 + 4*100 + \dots + n*100$$

```

Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}

I      1      2      3      4      5.....n
J      1      2      3      4      5.....n
K      1*100  2*100  3*100  4*100  5*100.....n*100

```

$$\begin{aligned}
& 1*100 + 2*100 + 3*100 + 4*100 + \dots + n*100 \\
& 100(1+2+3+4+\dots+n)
\end{aligned}$$

```

Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
            for(k=1;k<=100;k++){
                print("Hello");
            }
        }
    }
}

I      1      2      3      4      5.....n
J      1      2      3      4      5.....n
K      1*100  2*100  3*100  4*100  5*100.....n

```

$$\begin{aligned}
& 1*100 + 2*100 + 3*100 + 4*100 + \dots + n*100 \\
& 100(1+2+3+4+\dots+n) \\
& n(n+1)/2 \text{ ie } O(n^2)
\end{aligned}$$

```
Function(){
    int i,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

i

J

K

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I        1  
J  
K

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}

I      1
J      1
K
```

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<=  $i^2$ ;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1
J	1
K	$1 * n / 2$

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2
J	1	
K	1*n/2	

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2
J	1	4
K	1*n/2	

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I        1        2

J        1        4

K        1\*n/2    4\*n/2

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2	3
J	1	4	
K	1*n/2	4*n/2	

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2	3
J	1	4	9
K	1*n/2	4*n/2	

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I        1        2        3

J        1        4        9

K        1\*n/2    4\*n/2    9\*n/2

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2	3	4
J	1	4	9	
K	1*n/2	4*n/2	9*n/2	

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2	3	4
J	1	4	9	16
K	1*n/2	4*n/2	9*n/2	

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2	3	4
J	1	4	9	16
K	1*n/2	4*n/2	9*n/2	16*n/2

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2	3	4	5
J	1	4	9	16	
K	1*n/2	4*n/2	9*n/2	16*n/2	

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2	3	4	5
J	1	4	9	16	25
K	1*n/2	4*n/2	9*n/2	16*n/2	25*n/2

```
Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}
```

I	1	2	3	4	5
J	1	4	9	16	25
K	1*n/2	4*n/2	9*n/2	16*n/2	25*n/2

```

Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}

I      1      2      3      4      5.....n
J      1      4      9      16     25
K      1*n/2  4*n/2  9*n/2  16*n/2 25*n/2

```

```

Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}

I      1      2      3      4      5.....n
J      1      4      9      16     25.....n2
K      1*n/2  4*n/2  9*n/2  16*n/2 25*n/2

```

```

Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}

I      1      2      3      4      5.....n
J      1      4      9      16     25.....n2
K      1*n/2  4*n/2  9*n/2  16*n/2 25*n/2.....n2*n/2

```

```

Function(){
    int I,j,k,n;
    for(i=1;i<=n;i++){
        for(j=1;j<= i2;j++){
            for(k=1;k<=n/2;k++){
                printf("Hello");
            }
        }
    }
}

I      1      2      3      4      5.....n
J      1      4      9      16     25..... n2
K      1*n/2  4*n/2  9*n/2  16*n/2 25*n/2.....n2*n/2
1*n/2  4*n/2  9*n/2  16*n/2 25*n/2.....n2*n/2
n/2(12+22 + 32..... n2)

```

$$n/2((n(n+1)(2n+1)/6)$$

$$O(n^4)$$

```
Function(){  
    for(i=1;i<n;i=i*2)  
        printf("hello");  
}
```

i=1	2	4	8	16	n
$2^0$	$2^1$	$2^2$	$2^3$	$2^4$	$2^k$
$2^k = n$					

$$K = \log(n)$$

$$O(\log_2 n)$$

```
Function(){  
    int i,j,k;  
    for(i=n/2;i<=n;i++)  
        for(j=1;j<=n/2;j++)  
            for(k=1;k<=n;k=k*2)  
                printf("hello");  
}
```

```
Function(){
    int i,j,k;
    n/2        for(i=n/2;i<=n;i++)
    n/2        for(j=1;j<=n/2;j++)
    Log2 n        for(k=1;k<=n;k=k*2)
                    printf("hello");
}
n/2 *n/2*log2 n
O(n2 log2 n)
```

```
Function(){  
    int l,j,k;  
    for(i=n/2;i<=n;i++)  
        for(j=1;j<=n;j=2*j)  
            for(k=1;k<=n;k=k*2)  
                printf("Hello");  
}
```

```
Function(){
    int l,j,k;
n/2    for(i=n/2;i<=n;i++)
Log2 n        for(j=1;j<=n;j=2*j)
Log2 n        for(k=1;k<=n;k=k*2)
                printf("Hello");
}
n/2* (Log2 n)2
O(n* (Log2 n)2)
```

```
Function(n){  
    if(n>1)  
        return(function(n-1))  
}
```

```
Function(n){  
    if(n>1)          T(1)  
        return(function(n-1))  
}  
      
```

```
Function(n){  
    if(n>1)          T(1)  
        return(function(n-1)) T(n-1)  
}  
}
```

```
Function(n){  
    if(n>1)          T(1)  
        return(function(n-1))  T(n-1)  
}  
  

$$T(n) = 1 + T(n-1) \quad \text{eq(1)}$$

```

```
Function(n){  
    if(n>1)                                T(1)  
        return(function(n-1))                T(n-1)  
}  
T(n)=1+T(n-1);  n>1                      eq(1)
```

$$T(1) = 1+T(1-1); \quad n=1$$

$$T(1) = 1+T(0);$$

$$T(1) = 1$$

$$T(n)=1+T(n-1) \quad \text{eq(1)}$$

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
Function(n){  
    if(n>1)          T(1)  
        return(function(n-1))   T(n-1)  
}  
T(n)=1+T(n-1)      eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

$T(n-1)=1+T(n-2)$  eq(2)

```
Function(n){  
    if(n>1)          T(1)  
        return(function(n-1))   T(n-1)  
}  
T(n)=1+T(n-1)      eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

$$T(n-1)=1+T(n-2) \quad \text{eq}(2)$$

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
Function(n){  
    if(n>1)          T(1)  
        return(function(n-1))   T(n-1)  
}  
T(n)=1+T(n-1)      eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
T(n-1)=1+T(n-2)      eq(2)
```

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
T(n-2)=1+T(n-3)      eq(3)
```

```
Function(n){  
    if(n>1)          T(1)  
        return(function(n-1))  T(n-1)  
}  
T(n)=1+T(n-1)      eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
T(n-1)=1+T(n-2)      eq(2)
```

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
T(n-2)=1+T(n-3)      eq(3)
```

Backward substitution

```
Function(n){  
    if(n>1)                                T(1)  
        return(function(n-1))                T(n-1)  
}  
T(n)=1+T(n-1)                                eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

$T(n-1)=1+T(n-2)$  eq(2)

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

$T(n-2)=1+T(n-3)$  eq(3)

Backward substitution

Substitute eq(2) in 1  $T(n) = 1 + (1 + T(n-2)) = 2 + T(n-2)$

```
Function(n){  
    if(n>1)                                T(1)  
        return(function(n-1))                T(n-1)  
}  
T(n)=1+T(n-1)                                eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
T(n-1)=1+T(n-2)                                eq(2)
```

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
T(n-2)=1+T(n-3)                                eq(3)
```

Backward substitution

Substitute eq(2) in 1  $T(n) = 1+(1+T(n-2)) = 2+T(n-2)$

Substitute eq(3) in 2  $T(n)=2+(1+T(n-3)) = 3+T(n-3)$

```
Function(n){  
    if(n>1)                                T(1)  
    return(function(n-1))                    T(n-1)  
}  
T(n)=1+T(n-1)                      eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
T(n-1)=1+T(n-2)                      eq(2)
```

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
T(n-2)=1+T(n-3)                      eq(3)
```

Backward substitution

Substitute eq(2) in 1  $T(n) = 1 + (1 + T(n-2)) = 2 + T(n-2)$

Substitute eq(3) in 2  $T(n) = 2 + (1 + T(n-3)) = 3 + T(n-3)$

```
K+T(n-k)                            eq(4)
```

```
Function(n){  
    if(n>1)                                T(1)  
    return(function(n-1))                    T(n-1)  
}  
T(n)=1+T(n-1)                      eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
T(n-1)=1+T(n-2)                      eq(2)
```

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
T(n-2)=1+T(n-3)                      eq(3)
```

Backward substitution

Substitute eq(2) in 1  $T(n) = 1 + (1 + T(n-2)) = 2 + T(n-2)$

Substitute eq(3) in 2  $T(n) = 2 + (1 + T(n-3)) = 3 + T(n-3)$

```
K+T(n-k)                            eq(4)
```

When  $n-k=1$  loop stop so  $k=n-1$  substitute in eq(4)

```
Function(n){  
    if(n>1)                                T(1)  
    return(function(n-1))                    T(n-1)  
}  
T(n)=1+T(n-1)                                eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
T(n-1)=1+T(n-2)                                eq(2)
```

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
T(n-2)=1+T(n-3)                                eq(3)
```

Backward substitution

Substitute eq(2) in 1  $T(n) = 1 + (1 + T(n-2)) = 2 + T(n-2)$

Substitute eq(3) in 2  $T(n)=2+(1+T(n-3))=3+T(n-3)$

$K+T(n-k)$

When  $n-k=1$  loop stop so  $k=n-1$  substitute in eq(4)

$n-1+T(n-(n-1)))$

```
Function(n){  
    if(n>1)          T(1)  
    return(function(n-1))   T(n-1)  
}  
T(n)=1+T(n-1)           eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
T(n-1)=1+T(n-2)           eq(2)
```

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
T(n-2)=1+T(n-3)           eq(3)
```

Backward substitution

Substitute eq(2) in 1  $T(n) = 1 + (1 + T(n-2)) = 2 + T(n-2)$

Substitute eq(3) in 2  $T(n) = 2 + (1 + T(n-3)) = 3 + T(n-3)$

$K + T(n-k)$

When  $n-k=1$  loop stop so  $k=n-1$  substitute in eq(4)

$n-1 + T(n-(n-1)))$

$n$

```
Function(n){  
    if(n>1)          T(1)  
        return(function(n-1))   T(n-1)  
}  
T(n)=1+T(n-1)           eq(1)
```

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

```
T(n-1)=1+T(n-2)           eq(2)
```

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

```
T(n-2)=1+T(n-3)           eq(3)
```

Backward substitution

Substitute eq(2) in 1  $T(n) = 1 + (1 + T(n-2)) = 2 + T(n-2)$

Substitute eq(3) in 2  $T(n) = 2 + (1 + T(n-3)) = 3 + T(n-3)$

$K + T(n-k)$

When  $n-k=1$  loop stop so  $k=n-1$  substitute in eq(4)

$n - 1 + T(n-(n-1))$

$n$

$O(n)$

$$T(n) = n + T(n-1) \quad n > 1$$

$$T(1) = 1 + T(1-1) \quad n=1$$

$$T(n) = n + T(n-1) \quad \text{eq(1)}$$

To find  $T(n-1)$  substitute  $n-1$  to all  $n$  in eq(1)

$$T(n-1) = n-1 + T(n-2) \quad \text{eq(2)}$$

To find  $T(n-2)$  substitute  $n-2$  to all  $n$  in eq(1)

$$T(n-2) = n-2 + T(n-3) \quad \text{eq(3)}$$

Backward substitution

$$T(n) = n + T(n-1)$$

Substitute eq(2) in 1  $T(n) = n + (n-1) + T(n-2)$

Substitute eq(3) in 2  $T(n) = n + (n-1) + (n-2) + T(n-3) \dots + (n-k) + \underline{T(n-(k+1))}$  eq(4)

1

When  $n-(k+1)=1$  loop stop so  $k=n-2$  substitute in eq(4)

$$n-(k+1) = 1$$

$$n-k-1 = 1$$

$$K=n-2$$

$$T(1)$$

$$T(n) = n + (n-1) + (n-2) + T(n-3) \dots + (n-(n-2)) + T(n-((n-2)+1))$$

$$= n + (n-1) + (n-2) + T(n-3) \dots + 2 + 1$$

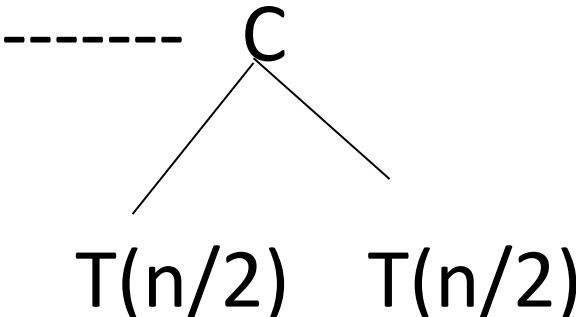
$$= n(n+1)/2$$

$$= O(n^2)$$

# Recursion Tree Method

- The recursion tree method is a way of solving recurrence relations
- In this method, a recurrence relation is converted into recursive trees
- Each node represents the cost incurred at various levels of recursion
- To find the total cost, costs of all levels are summed up

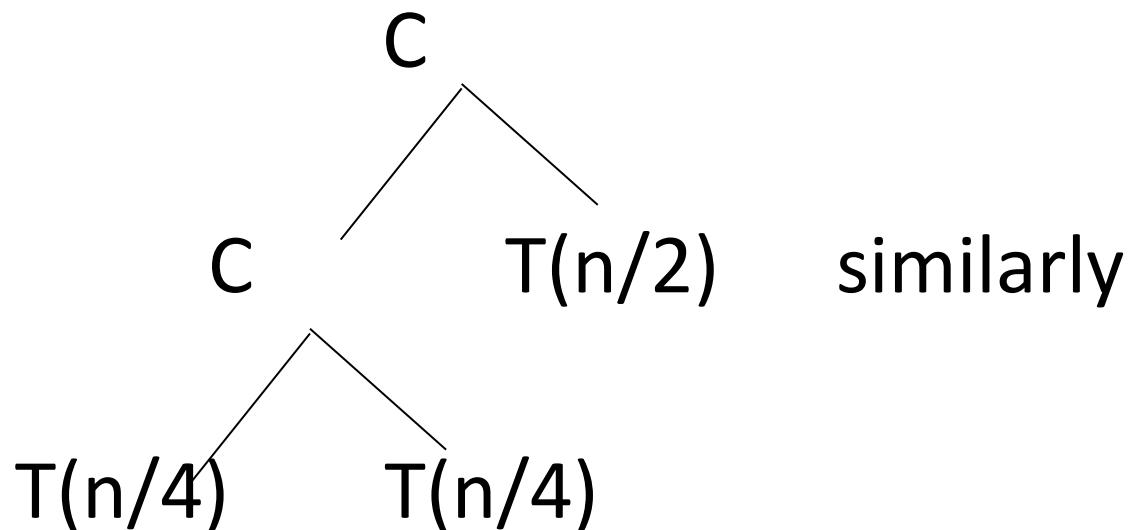
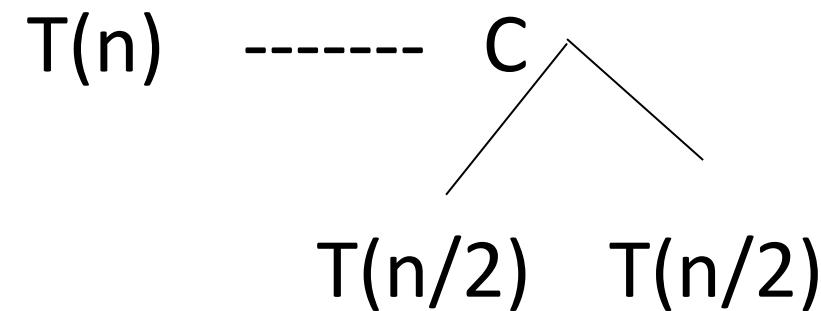
# Example 1

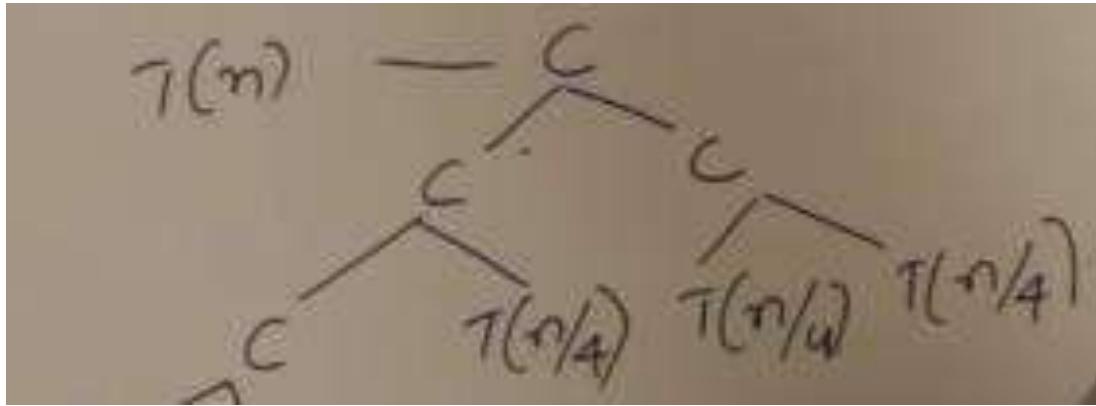
- $T(n) = 2T(n/2)+C \quad ; n>1$
- $= C \quad ; n=1$
- if I do  $C$  amount of work, I could produce 2 problems with smaller sets of inputs and each input size is reduced to  $n/2$ .
- $T(n)$  ----- 

```
graph TD; C((C)) --> T1["T(n/2)"]; C --> T2["T(n/2)"]
```

$$T(n) = \begin{cases} 2T(n/2)+C & ; n>1 \\ C & ; n=1 \end{cases}$$

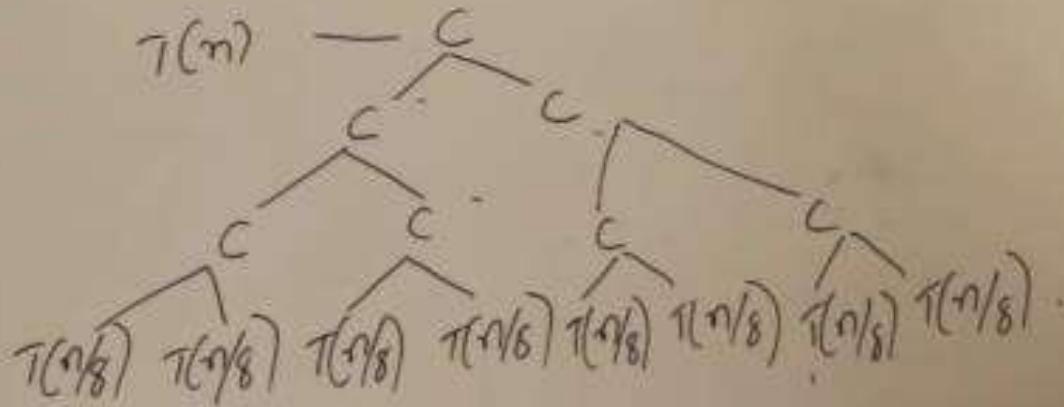
$$t(n/2) = \begin{cases} 2t(n/4)+c & ; n/2>1 \\ t(n/4) & ; n/2=1 \end{cases}$$





Recursion tree method:

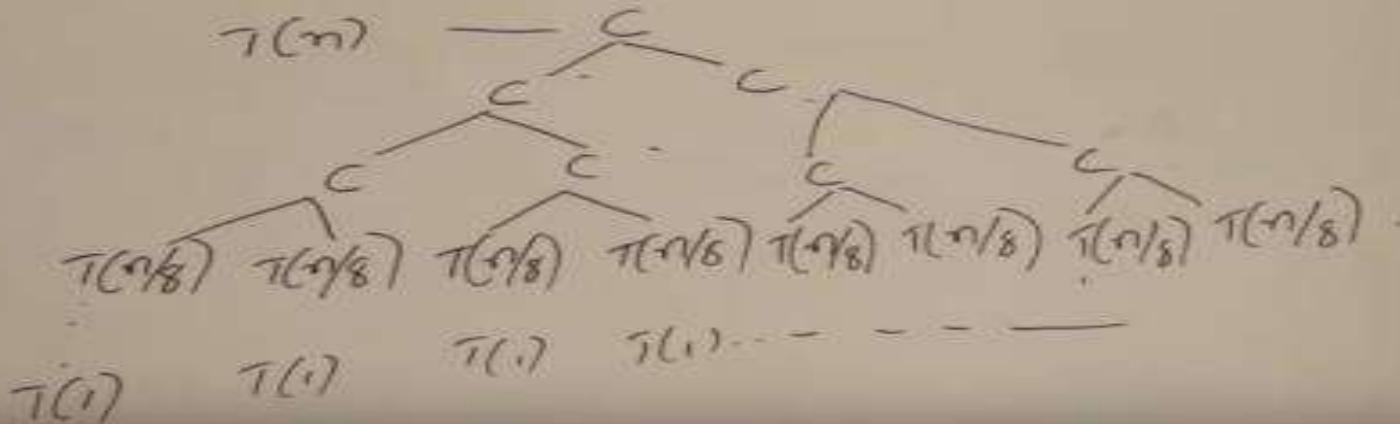
$$T(n) = \begin{cases} 2T(n/2) + C & ; n > 1 \\ = C & ; n = 1 \end{cases}$$



Recursion tree method :

$$T(n) = 2T(n/2) + C ; n > 1$$
$$= C ; n = 1$$

1C  
2C  
4C  
8C



$T(n/n) \dots \dots T(n/n)$

$nC$

$T(1) = T(n/n)$

$C + 2C + 4C + \dots NC$

$C(1+2+4+\dots n)$  assume  $n=2^k$  (for simplification)

$C(1+2+4+\dots 2^k) \Rightarrow C(1(2^{k+1}-1)) / (2-1) [G.P a(r^{n+1}-1)/(r-1)]$

$$\Rightarrow C(2^{k+1}-1)$$

$$\Rightarrow C(2^k \cdot 2^1 - 1)$$

$$\Rightarrow C(2n-1)$$

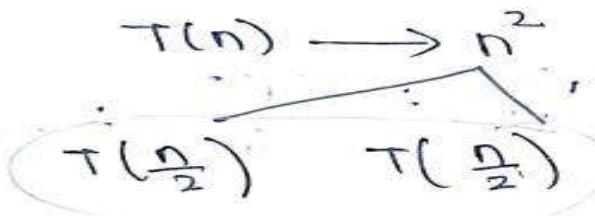
$$\Rightarrow O(n)$$

## Example 2

$$T(n) = 2T(n/2) + n^2 \quad ; \quad n > 1$$

$$= 1 \quad ; \quad n = 1$$

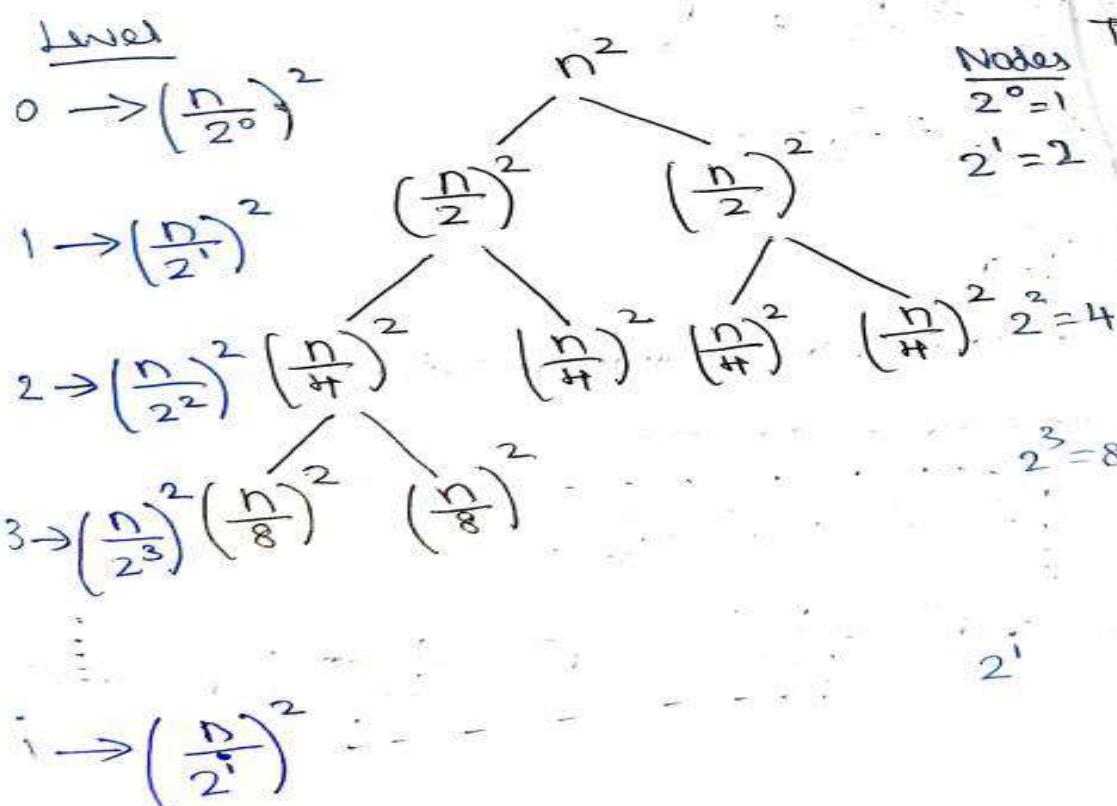
sol:



$$\begin{aligned} T\left(\frac{n}{2}\right) &= 2T\left(\frac{n}{2}\right) + \left(\frac{n}{2}\right)^2 \\ &= 2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2 \end{aligned}$$

$$\begin{aligned} T\left(\frac{n}{4}\right) &= 2T\left(\frac{n}{4}\right) + \left(\frac{n}{4}\right)^2 \\ &= 2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2 \end{aligned}$$

$$\begin{aligned} T\left(\frac{n}{8}\right) &= 2T\left(\frac{n}{8}\right) + \left(\frac{n}{8}\right)^2 \\ &= 2T\left(\frac{n}{16}\right) + \left(\frac{n}{8}\right)^2 \end{aligned}$$



Summation of each series

$$n^2 + 2\left(\frac{n}{2}\right)^2 + 4\left(\frac{n}{4}\right)^2 + 8\left(\frac{n}{8}\right)^2 + \dots$$

To find upper bound let's assume this series will go to infinite

$$= n^2 + 2\frac{n^2}{4} + 4\left(\frac{n^2}{16}\right) + 8\left(\frac{n^2}{64}\right) + \dots$$

$$= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \dots$$

$$= n^2 \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots\right)$$

$$\text{G.P} = \frac{a}{1-r}$$

$a \rightarrow$  start term,  $r \rightarrow$  common ratio

$$a + ar + ar^2 + ar^3 + \dots$$

$$= n^2 \left(1 + 1 \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{4} \dots\right)$$

$$= n^2 \left(\frac{1}{1-\frac{1}{2}}\right) = n^2 \left(\frac{1}{\frac{1}{2}}\right) = n^2 \left(\frac{1}{\frac{1}{2}}\right) = n^2 \left(1 \times \frac{2}{1}\right)$$

$$= n^2 (2)$$

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



# **BCSE202L & BCS202P- Data Structures and Algorithms**

**Dr. Priyanka N**

**Assistant Professor Senior Grade I**

**School of Computer Science & Engineering**

**VIT, Vellore.**

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the k <sup>th</sup> minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

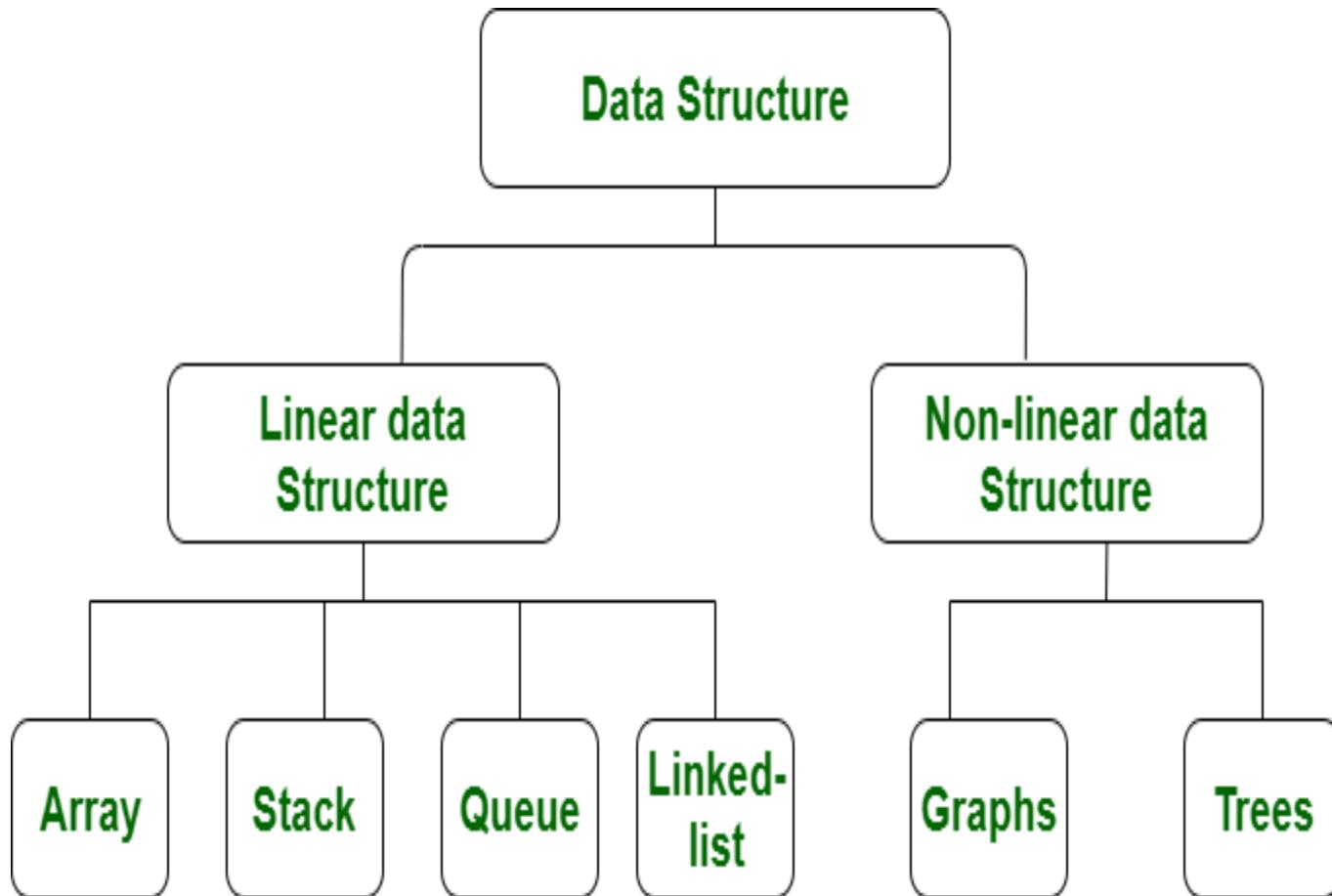
# **BCS202L- Data Structures and Algorithms**

## **Text Books:**

- I.** Mark A. Weiss, Data Structures & Algorithm Analysis in C++, 4 th Edition, 2013,Pearson Education.

## **Reference Books:**

- I.** Alfred V. Aho, Jeffrey D. Ullman and John E. Hopcroft, Data Structures and Algorithms,1983, Pearson Education.
- 2.** Horowitz, Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, 2008, 2<sup>nd</sup> Edition, Universities Press.
- 3.** Thomas H. Cormen, C.E. Leiserson, R L. Rivest and C. Stein, Introduction to Algorithms, 2009, 3<sup>rd</sup> Edition, MIT Press.



# Linear Data Structure

- Data are arranged in a linear sequence.
- Elements are placed or accessed in contiguous memory location.
- Objects (items) in the data structure can be traversed in a single run.
- Some examples are array, stack, queue.

# Stacks

- What is Stack?

- A stack is an ordered collection of homogeneous data element where the insertion and deletion take place at **one end**.
- It is called a Last-in-First-Out (LIFO)collection.
- It means, the item last entered will be first out.

# Examples



Stack of Books



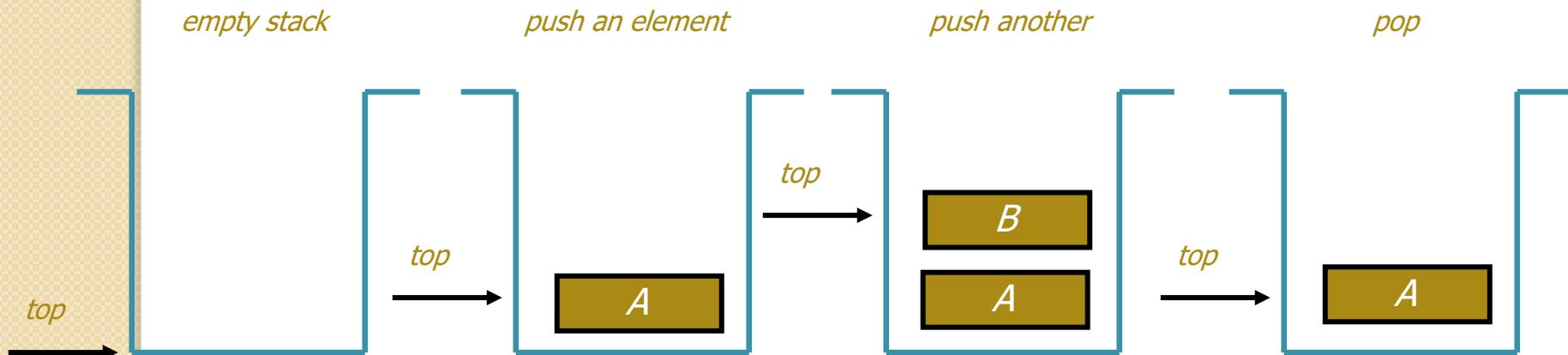
Plates in a tray

# Stacks Operations

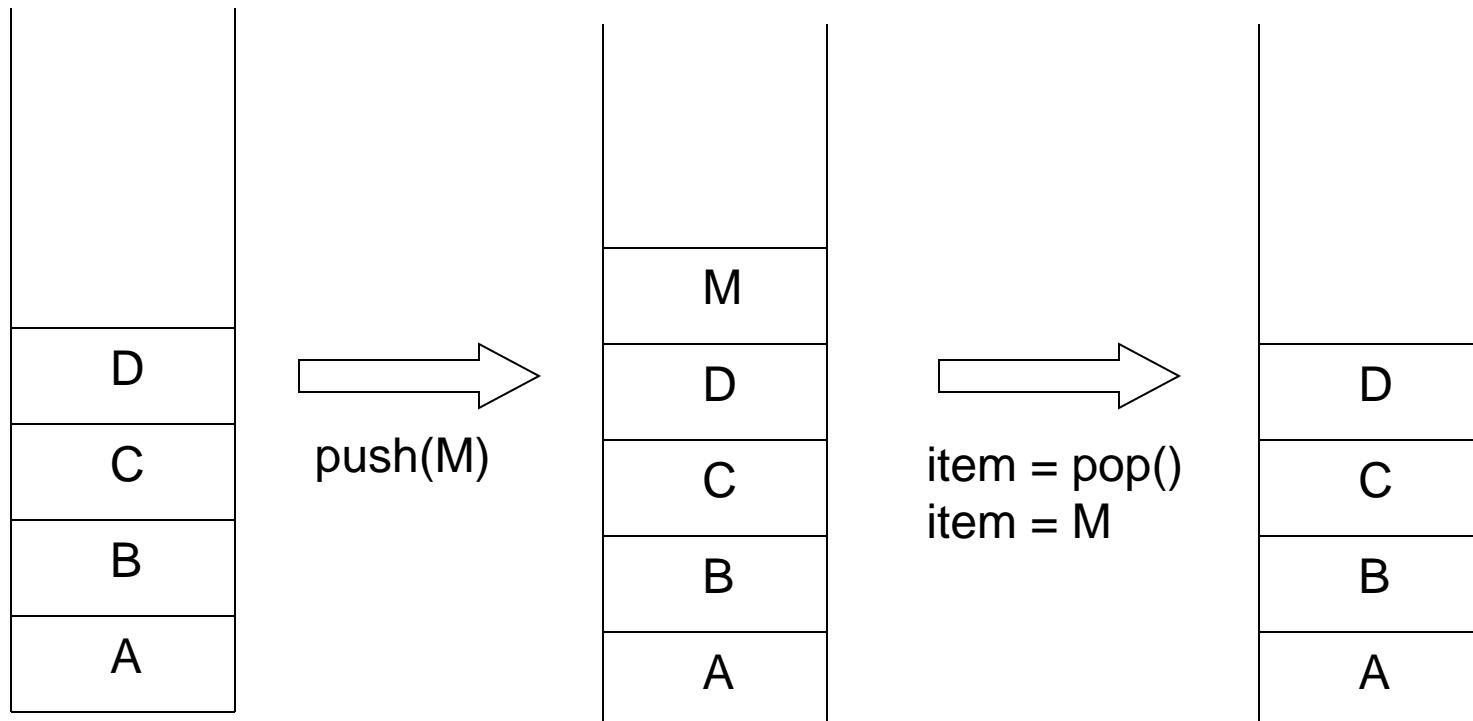
- **What can we do with a stack?**
  - **push** – operation to place an item on the stack
  - **pop** – operation to look at the item on top of the stack and remove it
  - Traversal- Displaying the items on the stack
  - is Full - To know whether the stack is full or not
  - is Empty - To know whether the stack is empty or not

# Push and Pop

- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



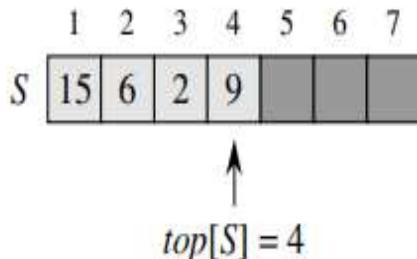
# Stack – push and pop example



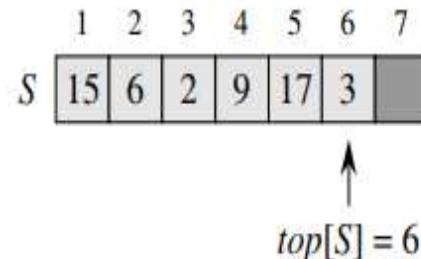
# Implementing a Stack

- Ways to implement a stack
  - Array
  - Linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

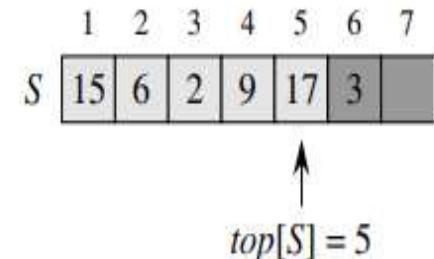
# Array Implementation of Stack



(a)



(b)



(c)

**Example :** An array implementation of a stack  $S$ . Stack elements appear only in the lightly shaded positions. **(a)** Stack  $S$  has 4 elements. The top element is 9. **(b)** Stack  $S$  after the calls  $PUSH(S, 17)$  and  $PUSH(S, 3)$ . **(c)** Stack  $S$  after the call  $POP(S)$  has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

# Implementing Stacks: Array

- **Advantages**
  - Best performance
- **Disadvantage**
  - Fixed size
- **Basic implementation**
  - Initially empty array
  - Field to record where the next data gets placed into
  - if array is full(stack Overflow), push() returns false
    - otherwise adds it into the correct spot
  - if array is empty(Stack Underflow), pop() returns null
    - otherwise removes the next(top) item in the stack

# Stack Class (Array based)

Variable top;

Variable MAX;

variable stack[MAX];

top=-1;

max=4;

Stackfull()

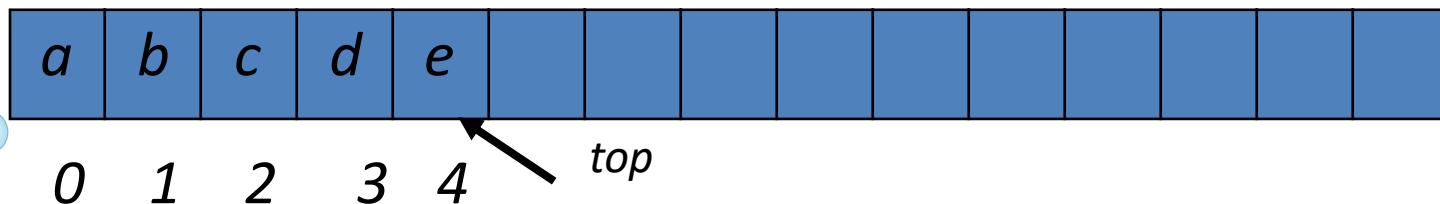
empty()

push(int value)

pop()

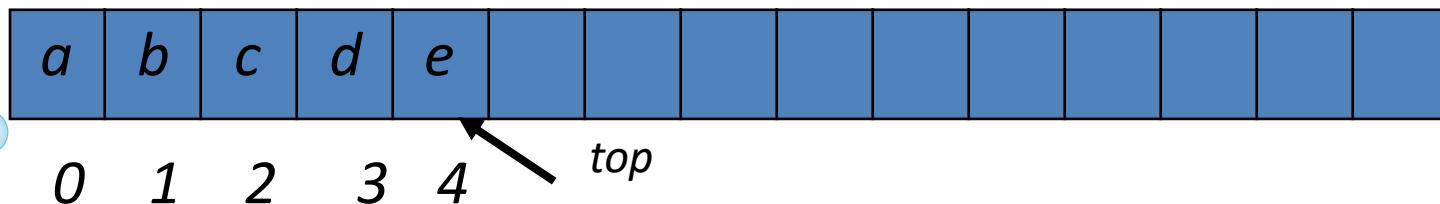
display()

# Push Operation



```
void push(element item)
{/* add an item to the global stack */
    if (top >= MAX_STACK_SIZE - 1)
        StackFull();
    /* add at stack top */
    stack[++top] = item;
}
```

# Pop Operation



```
Void pop()
{ /* Deletes an item from the stack*/
    if (top == -1)
        return StackEmpty();
    return stack[top--];
}
```

# Peek Operation



Void peek()

```
{ /* returns the top item from the stack*/  
if (top == -1)  
    return StackEmpty();  
return stack[top];  
}
```

# Remaining Methods (array based)

```
bool Stackempty()
{
    if (top== -1)
        return true;
    else
        return false;
}
```

```
Bool Stackfull()
{
    if (top== MAX_STACK_SIZE - 1)
        return true;
    else
        return false;
}
```

# Remaining Methods (array based)

```
void display()
{
    if (top==-1)
        return StackEmpty();
    else
    {
        cout<<"The elements in stack are:";
        for (int i=0;i<=top;i++)
            cout<<"\n"<<a[i];
    }
}
```

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10 // size of the stack

void push(int);
void pop();
void display();
int stack[SIZE], top = -1;

void main()
{
    int value, choice;
    clrscr();
    while(1)
    {
        printf("\n\n***** MENU *****\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
```

```
switch(choice)
{
    case 1: printf("Enter the value to be insert:");
              scanf("%d",&value);
              push(value);
              break;
    case 2: pop();
              break;
    case 3: display();
              break;
    case 4: exit(0);
    default: printf("\nWrong selection!!! Try again!!!");
}
}
```

```
void push(int value)

{
    if(top == SIZE-1)
        printf("\nStack is Full!!! Insertion is not possible!!!");

    else
    {
        top++;
        stack[top] = value;
        printf("\nInsertion success!!!");

    }
}
```

```
void pop()
{
    if(top == -1)
        printf("\nStack is Empty!!! Deletion is not possible!!!");
    else
    {
        printf("\nDeleted : %d", stack[top]);
        top--;
    }
}

void display()
{
    if(top == -1)
        printf("\nStack is Empty!!!");
    else
    {
        int i;
        printf("\nStack elements are:\n");
        for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
    }
}
```

# Stack Applications

- It is a very common data structure
- Stacks are used in conversion of infix to postfix expression.
- Stacks are also used in evaluation of postfix expression.
- Stacks are used to implement recursive procedures (eg:Towers of Hanoi).
- Stacks are used in compiler design.
- Memory Management in Operating System.

# **Applications of Stacks**

**Dr. Priyanka N**

**Assistant Professor Senior Grade I**

**School of Computer Science & Engineering  
VIT, Vellore.**

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# Stack(ADT)

- What is Stack?

- A stack is an ordered collection of homogeneous data element where the insertion and deletion take place at **one end**.
- It is called a Last-in-First-Out (LIFO)collection.
- It means, the item last entered will be first out.

# Examples



Stack of Books



Plates in a tray



Goods in a cargo

# Stacks Operations

- **What can we do with a stack?**
  - **push** – operation to place an item on the stack
  - **pop** – operation to look at the item on top of the stack and remove it
  - **Traversal**- Displaying the items on the stack
  - **is Full** - To know whether the stack is full or not
  - **is Empty** - To know whether the stack is empty or not
  -

# Stack Applications

- It is a very common data structure
- Stacks are used in conversion of infix to postfix expression.
- Stacks are also used in evaluation of postfix expression.
- Stacks are used to implement recursive procedures (eg:Towers of Hanoi).
- Stacks are used in compiler design.
- Memory Management in Operating System.

## • **Example I: Balancing Symbols**

- Make an empty Stack
- Read characters until end of file
- If the character is an opening symbol, push it onto the stack.
- If it is a closing symbol and the stack is empty, report an error, Otherwise pop the element in stack.
- If the symbol popped is not mapped with the corresponding opening symbol, then report an error.
- At end of file, if the stack is not empty, report an error.

# Example 2: Postfix Expressions

- Precedence of Operators - specifies how "tightly" it binds two expressions together.
- For Example:  $1+5*3=??$ 
  - 16 or 18 ?? - Ans: 16 because Multiplication ("\*") operator has a higher precedence than the addition ("+") operator.

## Typical Example:

- **4.99 \* 1.06 + 5.99 + 6.99 \* 1.06 = ???**
  - Multiply 4.99 and 1.06, saving this answer as **A<sub>1</sub>**,
  - Add 5.99 and A<sub>1</sub> and saves the result to **A<sub>1</sub>**,
  - Multiply 6.99 and 1.06, saving the answer in **A<sub>2</sub>**
  - Finish by adding **A<sub>1</sub>** and **A<sub>2</sub>**, leaving the final answer in **A<sub>1</sub>**
- **Sequence of Operation is written as**  
**4.99 1.06 \* 5.99 + 6.99 1.06 \* +**  
- **Post fix / Reverse Polish Notation**

# Post fix / Reverse Polish Notation

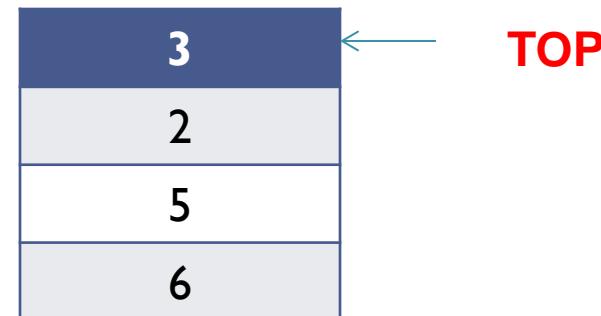
- **Evaluated easily by using stack**
  - When a number is seen, **it is pushed** onto the stack
  - When an operator is seen, the operator is applied to the **two numbers (symbols)** that are **popped** from the stack
  - Do the Operation and the **result is pushed** into the stack

# Post fix / Reverse Polish Notation

- For instance, the postfix expression

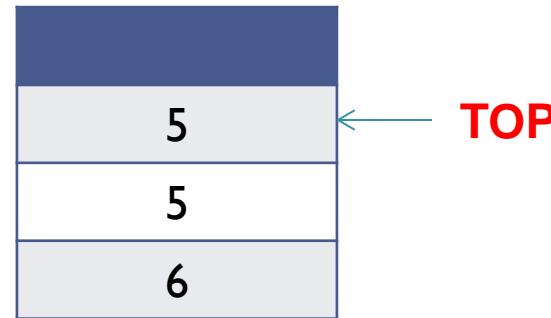
**6 5 2 3 + 8 \* + 3 + \***

- **First four symbols are placed on the stack**

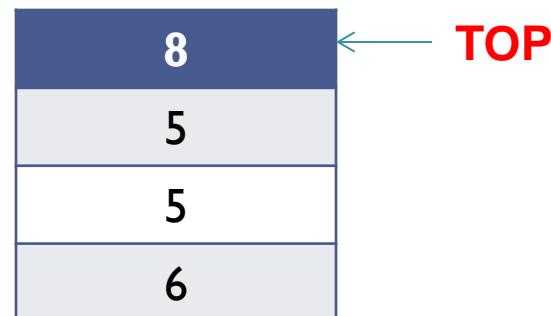


# Post fix / Reverse Polish Notation

- Next ‘+’ is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.

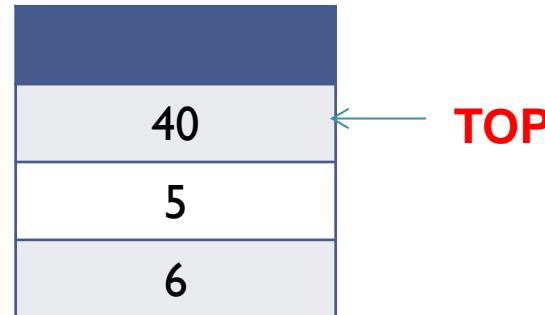


- Next ‘8’ is read, so it is pushed into the stack.

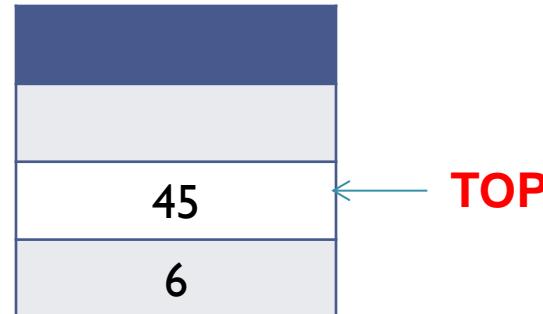


# Post fix / Reverse Polish Notation

- Next '\*' is read, so 8 and 5 are popped from the stack and their Product, 40, is pushed.

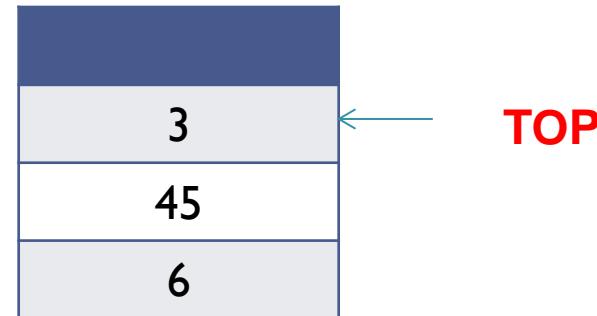


- Next '+' is read, so 40 and 5 are popped from the stack and their sum, 45, is pushed

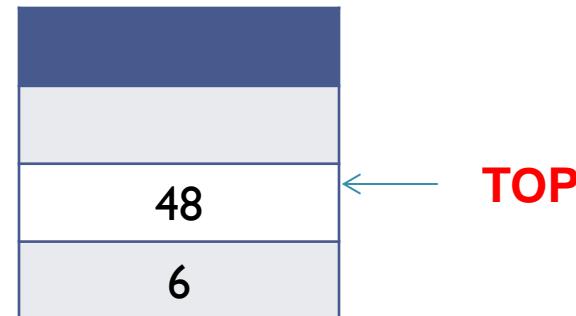


# Post fix / Reverse Polish Notation

- Next '3' is read, so it is pushed into the stack.

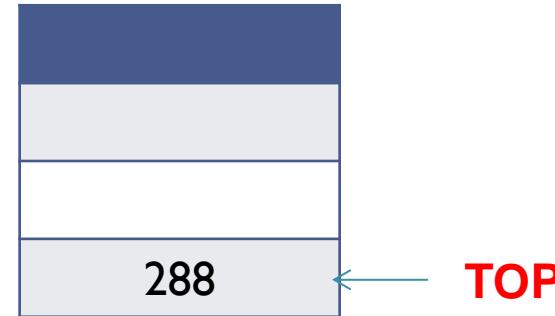


- Next '+' is read, so 45 and 3 are popped from the stack and their sum, 48, is pushed



# Post fix / Reverse Polish Notation

- Next '\*' is read, so 48 and 6 are popped from the stack and their Product, 288, is pushed.



- Time to evaluate a postfix expression is **O(N)**, because processing each element in the input consists of stack operations and therefore takes constant time.

# Example 3: Conversion of Infix to Postfix and Prefix Expression

- Infix to Postfix Conversion
  - Stack is used to convert an expression in standard form (otherwise known as infix) into postfix.
  - Example: **a + b \* c + ( d \* e + f ) \* g**

**Postfix Expression:** a b c \* + d e \* f + g \* +

# Rules

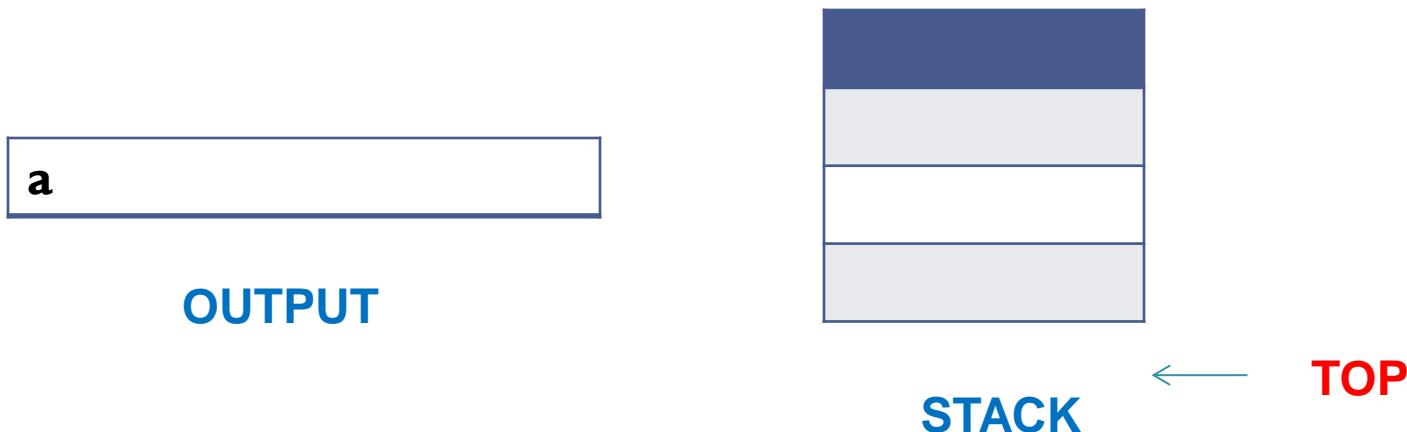
- When an operand is read, it is immediately placed onto the output.
- When an operator is seen, it is placed on the stack, stack represents pending operators.
- High precedence operator if present should be popped out before placing the current operator.(Except Parenthesis)

# Infix to Postfix Conversion

- For instance, the postfix expression

**a + b \* c + ( d \* e + f ) \* g**

- First the symbol “a” is read, it passed through the output.

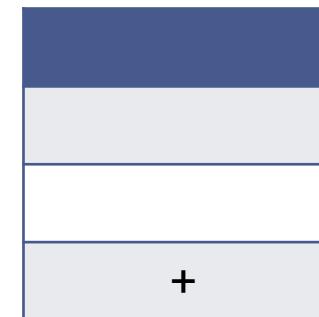


# Infix to Postfix Conversion

- Next “+” is read, it is pushed into the stack.



OUTPUT

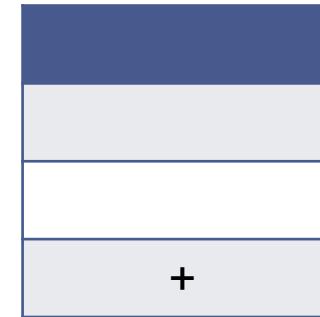


STACK

- Next “b” is read, it passed through the output.



OUTPUT



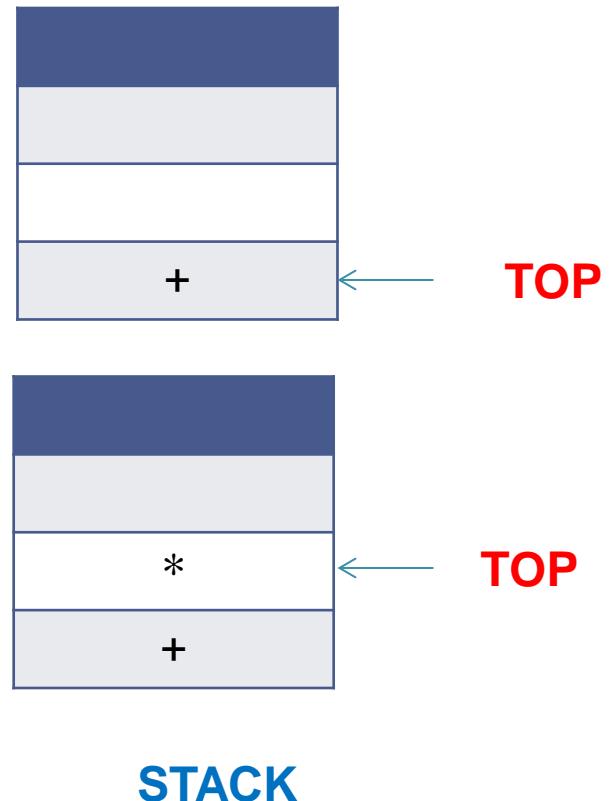
STACK

# Infix to Postfix Conversion

- Next “ \* ” is read, Check with top entry of the stack. Top entry “+” has lower precedence than \*. So push into the stack.

a b

OUTPUT

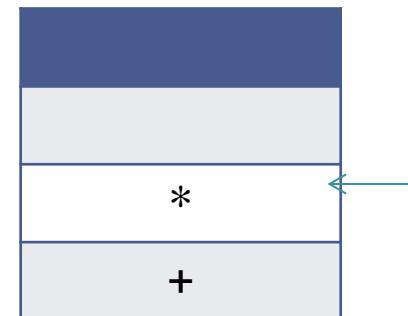


# Infix to Postfix Conversion

- Next “ c ” is read, it is passed to the output.

a b c

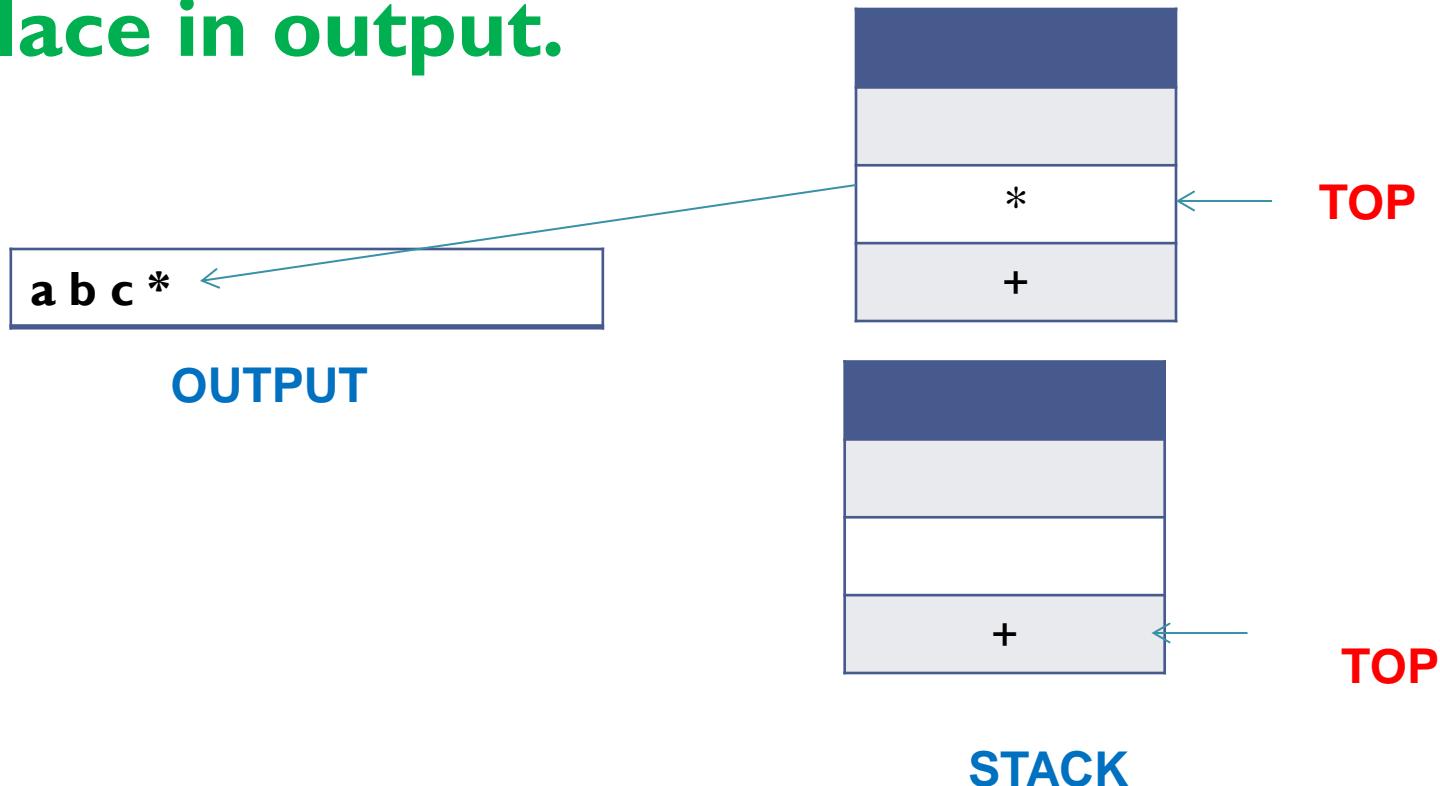
OUTPUT



STACK

# Infix to Postfix Conversion

- Next “ + ” is read, Check with the top entry of the stack. We have “ \* ”(higher precedence) – so pop it and place in output.

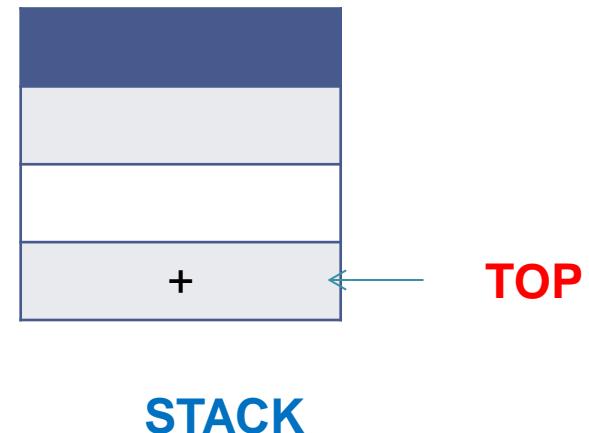


# Infix to Postfix Conversion

- Next “ + ” is read, Check with the top entry of the stack. We have “ + ”(which is not lower but equal priority) – place into the output

a b c \* +

OUTPUT

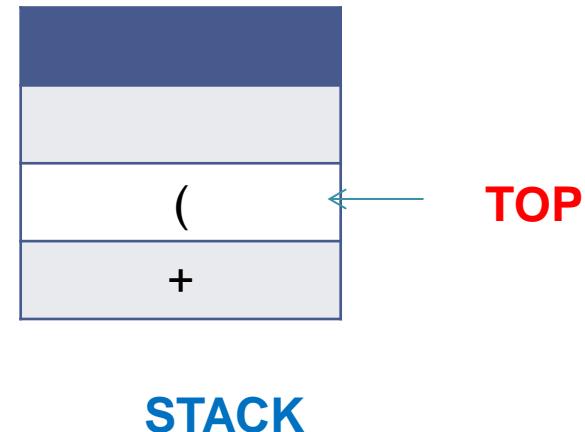


# Infix to Postfix Conversion

- Next “( ” is read, Check with the top entry of the stack. We have “+ ” where “(“ is **highest precedence** – push into the stack

a b c \* +

OUTPUT

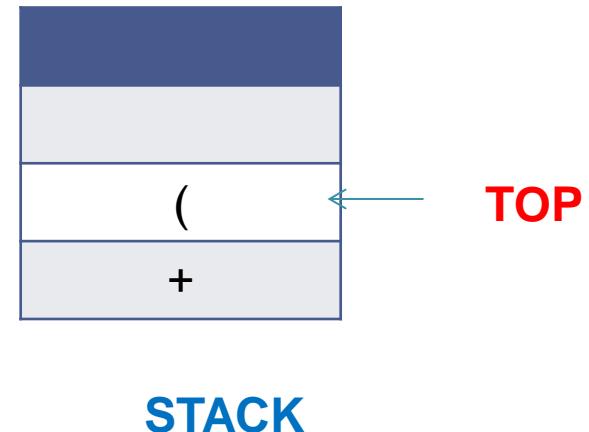


# Infix to Postfix Conversion

- Next “ d ” is read, it is passed into the output.

a b c \* + d

OUTPUT

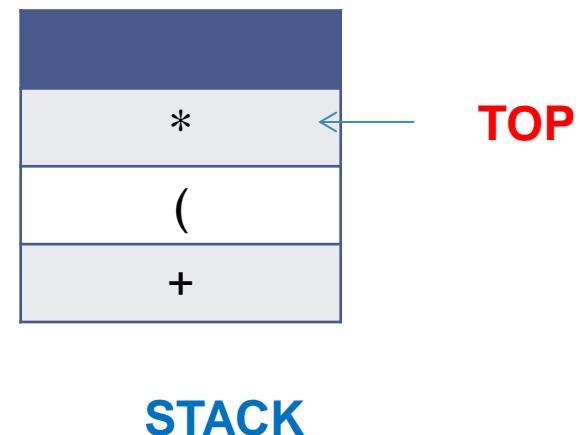


# Infix to Postfix Conversion

- Next “ \* ” is read, Check with the top entry of the stack. We have “ ( ” because of **highest precedence it is placed in stack).**
  - Note: Open parentheses do not get removed except when a closed parenthesis is being processed

a b c \* + d

OUTPUT

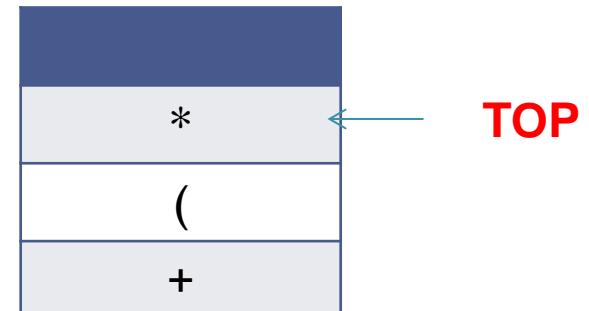


# Infix to Postfix Conversion

- Next “ e ” is read, it is passed into the output.

a b c \* + d e

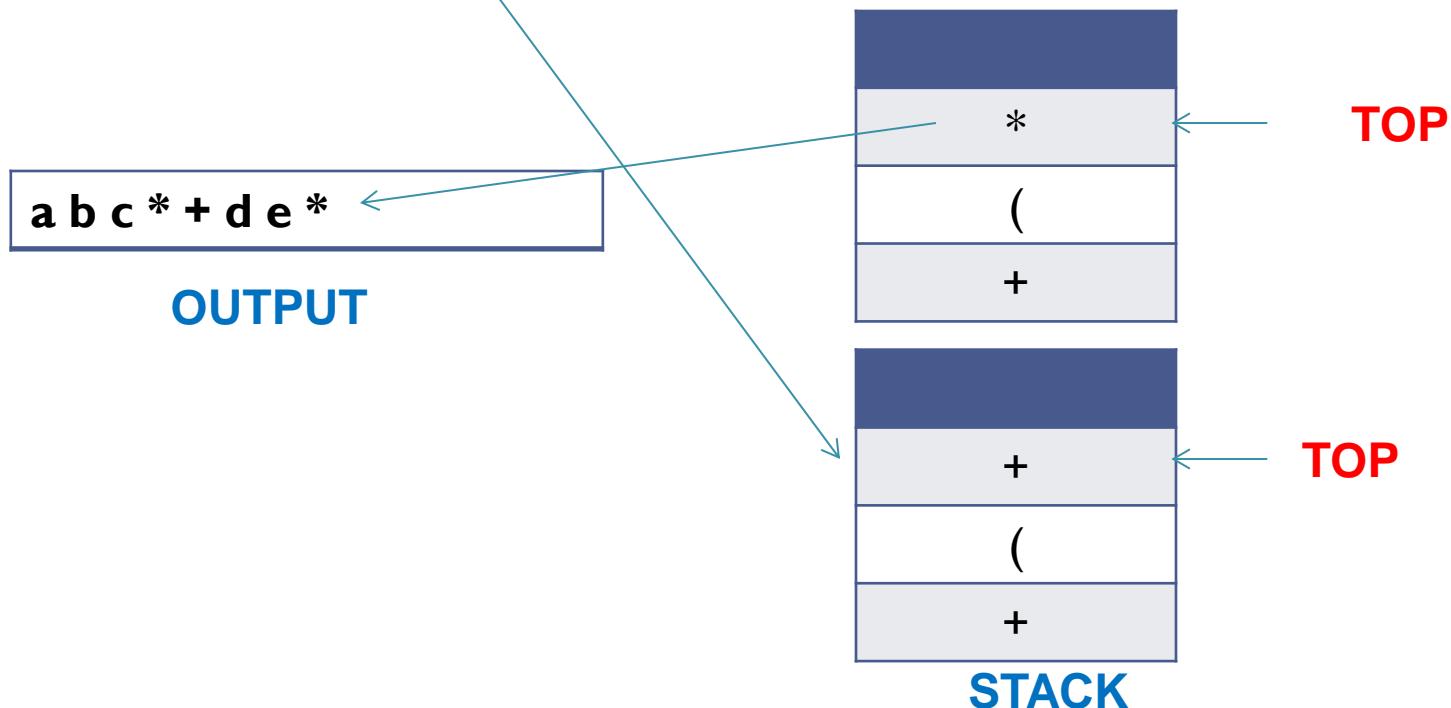
OUTPUT



STACK

# Infix to Postfix Conversion

- Next “ + ” is read, Check with the top entry of the stack. We have “ \* ”(highest precedence) – pop out and put into the output. And Push “+” into the stack

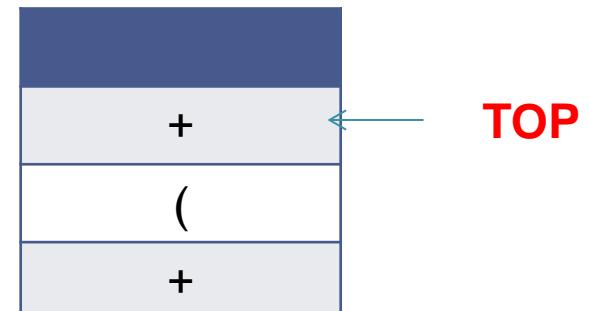


# Infix to Postfix Conversion

- Next “ f ” is read, it is passed into the output.

a b c \* + d e \* f

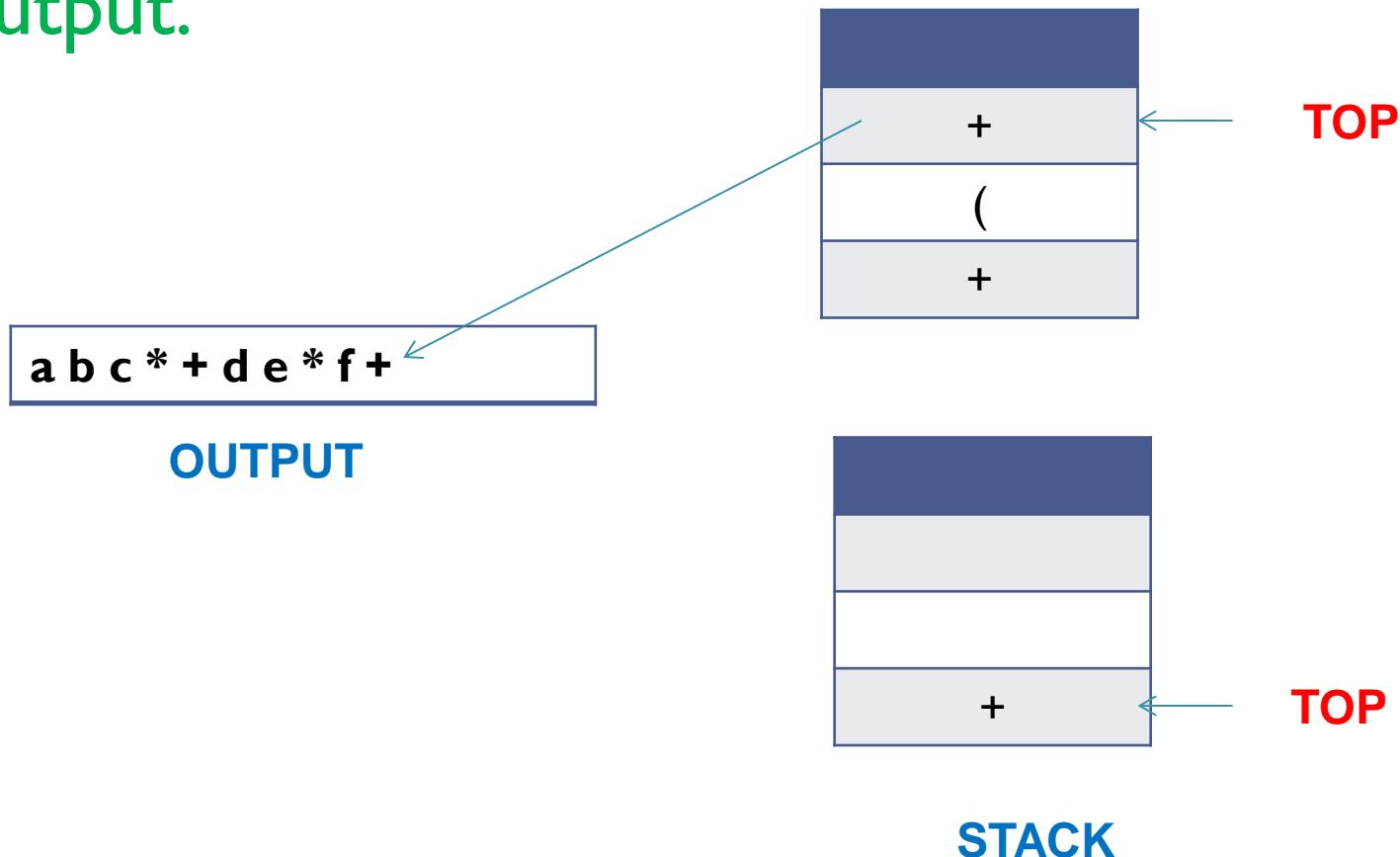
OUTPUT



STACK

# Infix to Postfix Conversion

- Next “ ) ” is read, stack is emptied back upto “ ( “. So “ + “ is passed into the output.

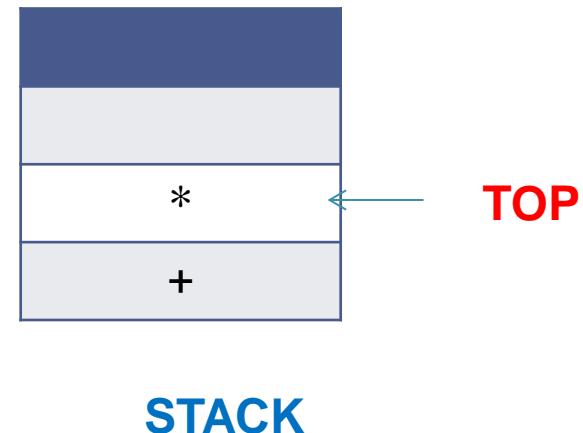


# Infix to Postfix Conversion

- Next “ \* ” is read, Check with the top entry of the stack. We have “ + ”(lower precedence) – push into the stack

a b c \* + d e \* f +

OUTPUT

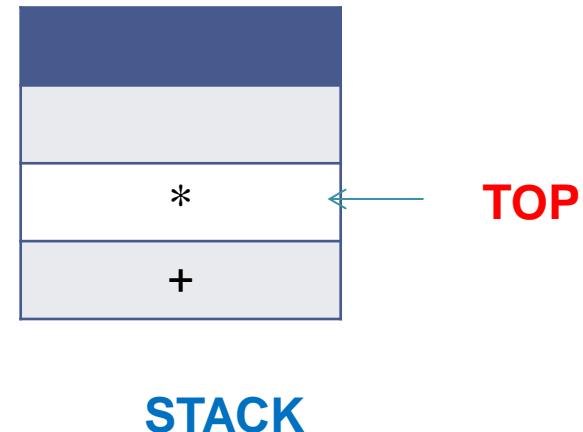


# Infix to Postfix Conversion

- Next “ g ” is read, it is passed into the output.

a b c \* + d e \* f + g

OUTPUT

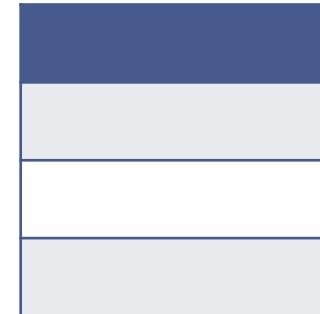


# Infix to Postfix Conversion

- Next input is empty , pop all the symbols on the stack until it is empty and move to output.

a b c \* + d e \* f + g \* +

OUTPUT



STACK

TOP

# Try Yourself( Infix to Postfix)

- $(a-b)/c*(d+e-f/g)$
- $(4+8)(6-5))/((3-2)(2+2))$



# **BCSE202L & BCS202P- Data Structures and Algorithms**

**Dr. Priyanka N**

**Assistant Professor Senior Grade I**

**School of Computer Science & Engineering**

**VIT, Vellore.**

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{th}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# **BCS202L- Data Structures and Algorithms**

## **Text Books:**

- I.** Mark A. Weiss, Data Structures & Algorithm Analysis in C++, 4 th Edition, 2013,Pearson Education.

## **Reference Books:**

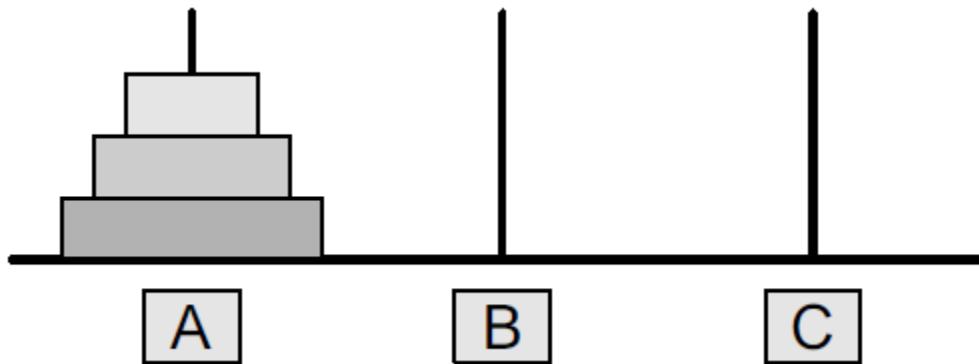
- I.** Alfred V. Aho, Jeffrey D. Ullman and John E. Hopcroft, Data Structures and Algorithms,1983, Pearson Education.
- 2.** Horowitz, Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, 2008, 2<sup>nd</sup> Edition, Universities Press.
- 3.** Thomas H. Cormen, C.E. Leiserson, R L. Rivest and C. Stein, Introduction to Algorithms, 2009, 3<sup>rd</sup> Edition, MIT Press.

# Tower of Hanoi

- The tower of Hanoi is one of the main applications of recursion
- It says, ‘if you can solve  $n-1$  cases, then you can easily solve the  $n$ th case’

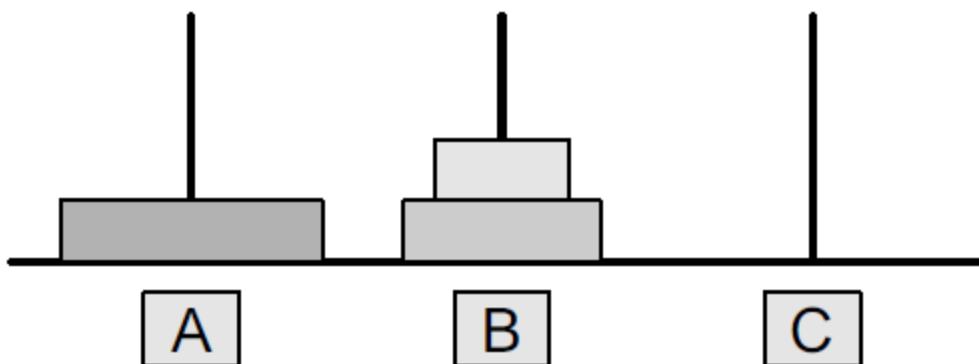
# Tower of Hanoi

- Three rings mounted on pole A
- The problem is to move all these rings from pole A to pole C while maintaining the same order
- The main issue is that the smaller disk must always come above the larger disk



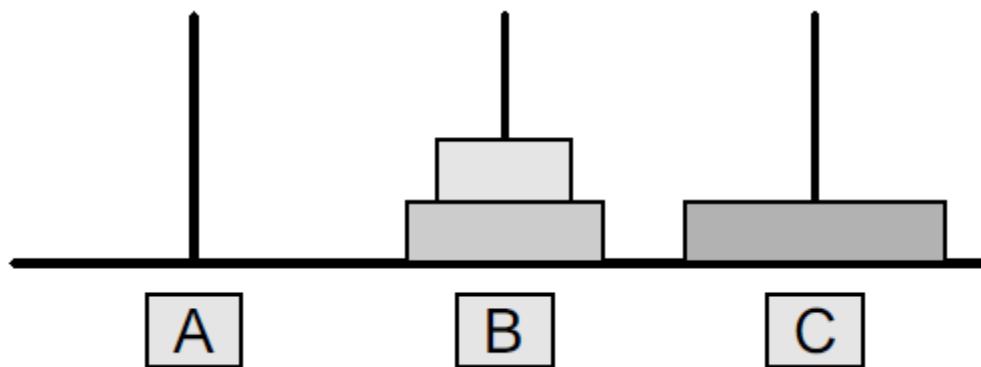
# Tower of Hanoi

- We will be doing this using a spare pole
- In this case, A is the source pole, C is the destination pole, and B is the spare pole
- To transfer all three rings from A to C, we will first shift the upper two rings ( $n-1$  rings) from the source pole to the spare pole
- We move the first two rings from pole A to B



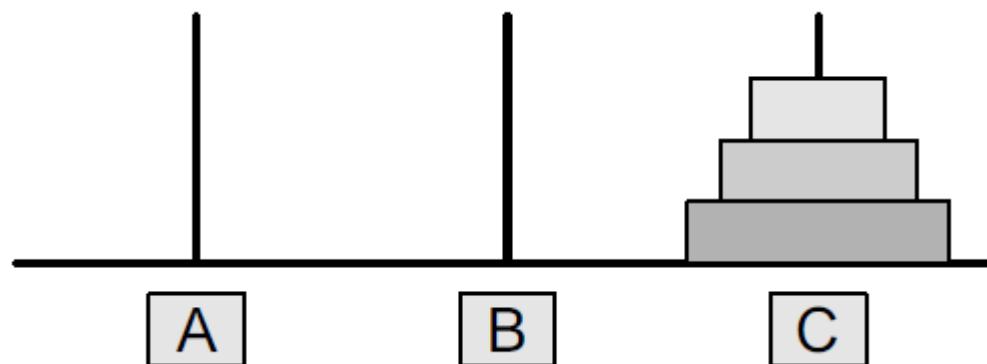
# Tower of Hanoi

- Now that  $n-1$  rings have been removed from pole A, the nth ring can be easily moved from the source pole (A) to the destination pole (C)



# Tower of Hanoi

- The final step is to move the  $n-1$  rings from the spare pole (B) to the destination pole (C)



# Tower of Hanoi

- To summarize, the solution to our problem of moving  $n$  rings from A to C using B as spare can be given as:

**Base case:** if  $n=1$

- Move the ring from A to C using B as spare

**Recursive case:**

- Move  $n - 1$  rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move  $n - 1$  rings from B to C using A as spare

# Program code

```
#include <stdio.h>

void towers(int, char, char, char);

int main()
{
    int num;
    printf("Enter the number of disks :");
    scanf("%d", &num);
    printf("The sequence of moves involved in the Tower of Hanoi are :\n");
    towers(num, 'A', 'C', 'B');
    return 0;
}

void towers(int num, char source, char destination, char spare)
{
    if (num == 1)
    {
        printf("\nMove disk 1 from pole %c to pole %c", source, destination);
        return;
    }
    // Recursively calling function twice
    towers(num - 1, source, spare, destination);
    printf("\nMove disk %d from pole %c to pole %c", num, source, destination);
    towers(num - 1, spare, destination, source);
}
```

Enter the number of disks : 3

The sequence of moves involved in the Tower of Hanoi are :

Move disk 1 from pole A to pole C

Move disk 2 from pole A to pole B

Move disk 1 from pole C to pole B

Move disk 3 from pole A to pole C

Move disk 1 from pole B to pole A

Move disk 2 from pole B to pole C

Move disk 1 from pole A to pole C

## Infix to Postfix Conversion

### Rules :-

- When an operand is read, it is immediately placed onto the output.
- When an operator is seen, it is placed on the stack,  
stack represents pending operators.
- High precedence operator if present should be popped out before placing the current operator.

$$① a+b*c + (d*e+f)*g$$

Infix character Scanned	Stack	Postfix expression
a	-	a
+	+	a
b	+	ab
*	+*	ab
c	+*	abc
+	+	abc*+
(	+()	abc*+
d	+()+	abc*+d
*	+(*)	abc*+d
e	+(*)+	abc*+de
+	+()	abc*+de*
f	+()	abc*+de*f
)	+	abc*+de*f+
*	+*	abc*+de*f*
g	+	abc*+de*f+g

Ans  $\rightarrow abc*+de*f+g*$

$$② (A - (B | C + (D \% E * F) | G) * H)$$

Infix character	Stack	Postfix
(	(	-
A	(	A
-	(-	A
(	(-()	A
B	(-()	AB
/	(-() /	AB
C	(-() /	ABC
+	(-() +	ABC /
(	(-() + (	ABC /
D	(-() + (	ABC / D
\%	(-() + ( \% )	ABC / D
E	(-() + ( \% )	ABC / D E
*	(-() + ( * )	ABC / D E \% .
F	(-() + ( * )	ABC / D E \% . F
)	(-() +	ABC / D E \% . F *
/	(-() + /	ABC / D E \% . F *

G	( - ( + /	A B C   D E * ) . F * G )
)	( -	A B C   D E * ) . F * G ) / +
*	( - *	A B C   D E * ) . F * G ) / +
H	( - *	A B C   D E * ) . F * G ) / + H
)	( - *)	A B C   D E * ) . F * G ) / + H * -

### Expression evaluation

→ Postfix evaluation

→ Prefix evaluation.

### POSTFIX EVALUATION

Rules :-

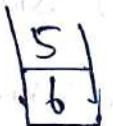
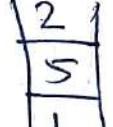
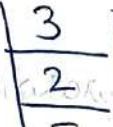
- When a number is seen, it is pushed onto the stack
- When an operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack
- Do the operation and the result is pushed into the stack.

## Example:-

Evaluate the postfix expression

$6 \ 5 \ 2 \ 3 + 8 * + 3 + *$

Sol:-

Symbol	Stack
6	Push "6" 
5	Push "5" 
2	Push "2" 
3	Push "3" 
+	Pop two operands Push (A) = 3 Push (B) = 2 The operation must be performed like $(B+A)$ . B $\rightarrow$ A order 

8

Push "8"

8
5
5
6

\*

$$5 * 8 = 40$$

40
5
6

+

$$5 + 40 = 45$$

45
6

3

Push "3"

3
45
6

+

$$45 + 3 = 48$$

48
6

\*

$$6 * 48 = 288$$

288
-----

Example 2: Evaluate the postfix expression

$$7 \ 5 \ 3 \ * \ 5 \ 1 \wedge \ 1 \ + \ 3 \ 2 \ - \ +$$

Sol:

Symbol	Stack
7	7
5	5 7
3	3 5 7
*	5 * 3 = 15 15 7
5	5 15 7
1	1 5 15 7
$\wedge$	$5 \wedge 1 = 5$ B A 5 15 7

$$\frac{15}{5} = 3$$

$$\boxed{\begin{array}{|c|}\hline 3 \\ \hline 7 \\ \hline \end{array}}$$

+

$$7 + 3 = 10$$

$$\boxed{\begin{array}{|c|}\hline 10 \\ \hline \end{array}}$$

3

$$\boxed{\begin{array}{|c|}\hline 3 \\ \hline 10 \\ \hline \end{array}}$$

2

$$\boxed{\begin{array}{|c|}\hline 2 \\ \hline 3 \\ \hline \end{array}} \quad \boxed{\begin{array}{|c|}\hline 10 \\ \hline \end{array}}$$

-

$$3 - 2 = 1$$

$$\boxed{\begin{array}{|c|}\hline 1 \\ \hline 10 \\ \hline \end{array}}$$

+

$$10 + 1 = 11$$

$$\boxed{\begin{array}{|c|}\hline 11 \\ \hline \end{array}}$$

Example 3 :- Evaluate the postfix expression

5 3 + 8 2 - \*

Sol:

Symbol	Stack
5	[5]
3	[3] [5]
+	[8]
8	[8] [8]
2	[2] [8] [8]
-	$8 - 2 = 6$ [6] [8]
*	$8 * 6 = 48$ [48]

## PREFIX EVALUATION

→ Reverse the expression

→  $\boxed{A} \downarrow \boxed{B}$  e.g.  $A+B$ . Evaluated by taking top operand A and then B  
Bottom operand B

Example 1 :- Evaluate the prefix notation

$+,-,* ,2,2,1,16,8,5$

Solution :

Reverse the expression  $\rightarrow 5,8,16,1,2,2,*,-,+,-$

Symbol	Stack
5	$ 5 $
8	$\frac{ 8 }{5}$
16	$\frac{ 16 }{\frac{8}{5}}$
/	$\frac{ 2 }{\frac{16}{8}}$
2,2	$\frac{\frac{2}{2}}{\frac{2}{5}}$
*	$\frac{4}{\frac{2}{5}}$
-	$\frac{2}{\frac{2}{5}}$
+	$7$

Example 2: Evaluate the prefix expression.

- , + , 2 , \* , 3 , 4 , 1 , 16 , ^ , 2 , 3

Sol:

Reverse  $\rightarrow 3, 2, \wedge, 16, 1, 4, 3, *, 2, +, -$

Symbol	Stack
3	$  3  $
2	$  2  $ $  3  $
$\wedge$	Here, A $\rightarrow$ 2, B $\rightarrow$ 3 It follows A $\rightarrow$ B order. $2^3 = 8$ $  8  $
16	$  16  $ $  8  $
/	$A/B = 16/8 = 2$ $  2  $
4	$  4  $ $  2  $
3	$  3  $ $  4  $ $  2  $
*	$3*4 = 12$ $  12  $ $  2  $

2

2
12
2

+

$$2+12=14$$

14
2

-

$$14-2=12$$

12
----

Example 3: Evaluate the prefix expression.

$$+9 * 2 \ 6$$

Sol:

$$\text{Reverse} \rightarrow 6 \ 2 * 9 +$$

Symbol	Stack
6	6
2	2
*	12
9	9
+	21

## Infix to Prefix conversion

Rules:

- (i) Reverse the infix expression
- (ii) Obtain the "nearly" postfix expression of the modified expression
- (iii) Reverse the postfix expression.

Example 1 :-  $(A + B \cdot C) * D + E \wedge 5$

Sol:-

Reverse  $\rightarrow 5 \wedge E + D * ) C \wedge B + A ($

Input	Stack	Output
5		5
$\wedge$	$\wedge$	5
E	$\wedge$	SE
+	+	SEA
D	+	SEAD
*	*+	SEAD
)	+*)	SEAD
C	+*)}	SEADC
$\wedge$	+*) $\wedge$	SEADC
B	+*) $\wedge$	SEADCB

+ +\*) + SEADCBA  
A +\*) + SEADCBA A  
( +\*) SEADCBA A +  
SEADCBA A + \* +  
DOME → +\* + A ABCD AES

### Rules:

- (i) First, reverse the infix expression given in the problem
  - (ii) Scan the expression from left to right
  - (iii) Whenever operands arrive, print them.
  - (iv) If the operator arrives & the stack is found to be empty, then simply push the operator into the stack.
  - (v) If the incoming operator has higher precedence than the top of the stack, push the incoming operators into the stack.
  - (vi) If the incoming operator has the same precedence with a top of the stack, push the incoming operator into the stack.
  - (vii) If the incoming operator has lower precedence than the top of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again & pop the operators from the stack till it finds the operators of a lower precedence or same precedence.
  - (viii) If the incoming operator has the same precedence with the top of the stack & the incoming operator is  $\wedge$ , then pop the top of the stack till the condition is true. If the condition is not true, push the  $\wedge$  operator.

## Rules (continued) :-

- (ix) When we reach the end of the expression, pop, and print all the operators from the top of the stack.
- (x) If the operator is ')', then push it into the stack.
- (xi) If the operator is '(', then pop all the operators from the stack till it finds ')' opening bracket in the stack.
- (xii) If the top of the stack is ')', push the operator on the stack.
- (xiii) At the end, reverse the output.

Example 2:  $K + L - M * N + (O \wedge P) * W \vee U \vee V * T + Q$

Sol: Reverse  $\rightarrow Q + T * V \vee U \vee W * P \wedge O (+ N * M - L + K)$

Symbol	Stack	Prefix
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTUV
/	+*/	QTUVW
W	+*/	QTUVW
*	+*/  *	QTUVW
)	+*/  *)	QTUVW
P	+*/  *)	QTUVWNP
^	+*/  *)^	QTUVWNP
O	+*/  *)^	QTUVWPO
(	+*/  *	QTUVWPOA
+	++	QTUVWPOA*/  *
N	++	QTUVWPOA*/  *N

\*

++\*

QTVUWPOΛ\*//N

M

++\*

QTVUWPOΛ\*//NM

-

++-

QTVUWPOΛ\*//NM\*

L

++-

QTVUWPOΛ\*//NM\*L

+

++-+

QTVUWPOΛ\*//NM\*L

K

++-+

QTVUWPOΛ\*//NM\*LK

QTVUWPOΛ\*//NM\*LK+-++

Reverse → ++-+KL\*MN\*//ΛOPWUVTQ

Eg 3:

Convert infix to prefix

$$A + B * C / D$$

Sol: Reverse  $\rightarrow D \mid C * B + A$

Symbol	Stack	prefix
D		D
/	/	DC
*	/*	DCB
B	/*B	DCB*
+	/*B+	DCB*/+
A	/*B+A	DCB*/+A

Reverse  $\rightarrow + A / * B C D$

$$\text{Eq 4: } A \wedge B * C / (D * E - F)$$

Sol:

$$\text{Reverse} \rightarrow F - E * D / C * B \wedge A$$

Symbol	Stack	Prefix
)	)	
F	)	F
-	)-	F
E	)-	FE
*	)-*	FE
D	)-*	FED
(		FED*-
/	/	FED*-
C	/	FED*-C
*	/*	FED*-C
B	/*	FED*-CB
$\wedge$	$/*\wedge$	$FED*-CB$
A	$/*\wedge$	$FED*-CBA$
		$FED*-CBA$
		$A \wedge */$

Reverse  $\rightarrow$

$/*\wedge ABC - *DEF$

Eq : 5: Convert infix to prefix

$$(A+B) * (C-D) + X^Y^Z * T$$

Solution: Reverse  $T * Z ^ Y ^ X + ) D - C ( * ) B + A ($

Symbol	Stack	Prefix
+		$T$
*	*	$T$
Z	*	$TZ$
^	*^	$TZ$
Y	*^	$TZY$
^	*^	$TZY^$
X	*^	$TZY^X$
+	+	$TZY^X^*$
)	+	$TZY^X^*$
D	+	$TZY^X^*D$
-	+)	$TZY^X^*D$
C	+)	$TZY^X^*DC$
(	+ $\boxed{-}$ $\rightarrow$ POP	$TZY^X^*DC-$
*	+*	$TZY^X^*DC-$
)	+*)	$TZY^X^*DC-$
B	+*)	$TZY^X^*DC-B$
+	+*)+	$TZY^X^*DC-B$
A	+*)+	$TZY^X^*DC-B A$

( | + \* ) + ( ) → POP

TZYΛXΛ\*DC-BA+

TZYΛXΛ\*DC-BA+ : \* +

Reverse → + \* + AB- CD \* X Λ Y Z T

# **QUEUE**

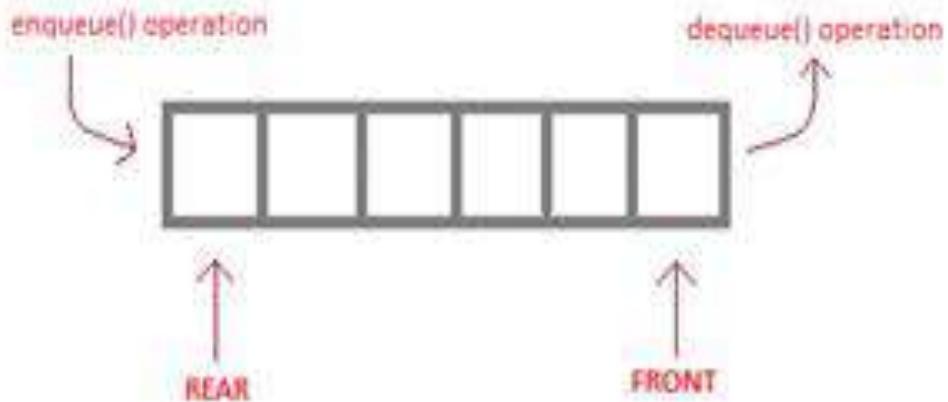


# Queue

- Ordered collection of homogeneous elements
- Non-primitive linear data structure.
- A new element is added at one end called rear end and the existing elements are deleted from the other end called front end.
- This mechanism is called First-In-First-Out (FIFO).

e.g. People standing in Queue for Movie Ticket

# Fig: Model of a Queue



`enqueue()` is the operation for adding an element into Queue.

`dequeue()` is the operation for removing an element from Queue.

## QUEUE DATA STRUCTURE

# Queue as ADT(Abstract data type.)

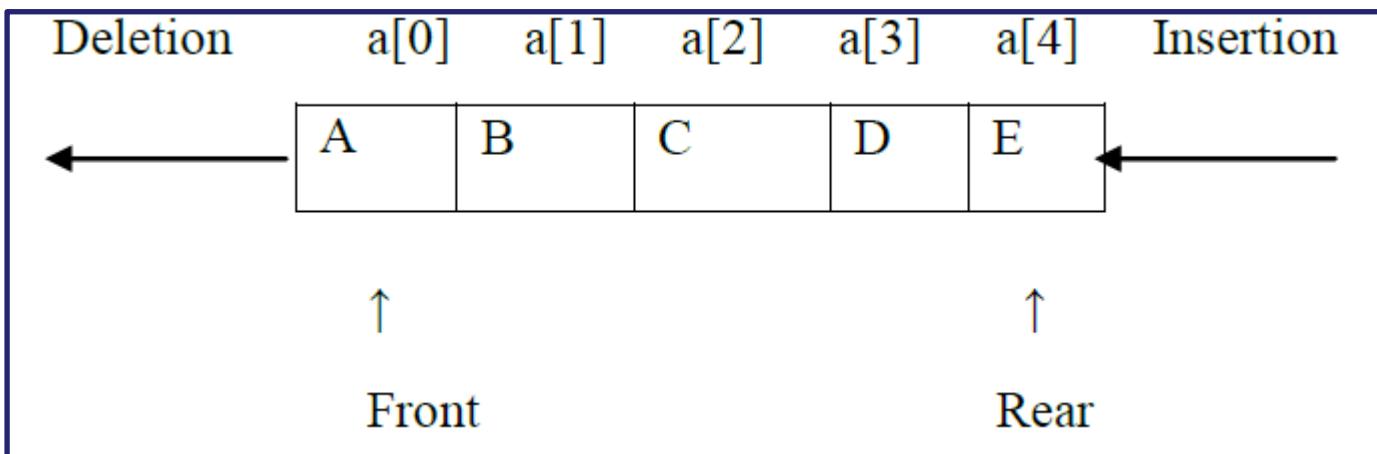
- Queue is a data structure which allows a programmer to insert an element at one end known as “Rear” and to delete an element at other end known as “Front”.
- Queue is an abstract data type because it not only allows storing elements but also allows to perform certain operation on these elements.

# Queue as ADT(Abstract data type.)

- These operations are as follows.
  - Initialize()
  - enqueue()
  - dequeue()
  - Isempty()
  - Isfull()
  - Display()
- Elements of queue:-
  - Front
  - Rear
  - array

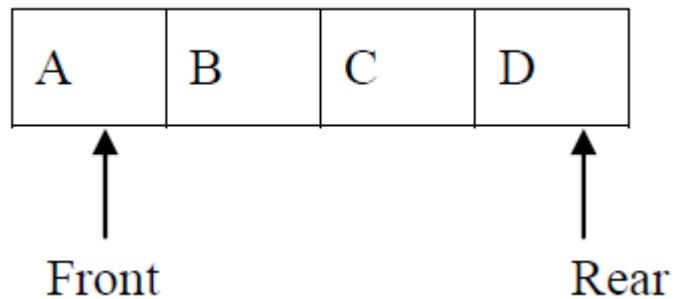
# Elements of queue:-

- **Front:** -
  - This end is used for deleting an element from a queue. Initially front end is set to -1. Front end is incremented by one when a new element has to be deleted from queue.
- **Rear:** -
  - This end is used for inserting an element in a queue. Initially rear end is set to -1. rear end is incremented by one when a new element has to be inserted in queue.



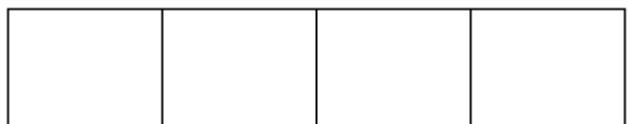
# 'Queue Full(Overflow)' Condition

- **Queue Full(Overflow):**
  - Inserting an element in a queue which is already full is known as Queue Full condition ( $\text{Rear} = \text{Max}-1$ ).
  - When the queue is fully occupied and `enqueue()` operation is called queue overflow occurs.
- **Example: Queue Full:**
  - Before inserting an element in queue 1<sup>st</sup> check whether space is available for new element in queue. This can be done by checking position of rear end. Array begins with 0<sub>th</sub> index position & ends with max-1 position. If numbers of elements in queue are equal to size of queue i.e. if rear end position is equal to max-1 then queue is said to be full. **Size of queue = 4**



# 'Queue Empty(Underflow)' Condition

- **Queue Empty:**
  - Deleting an element from queue which is already empty is known as Queue Empty condition ( $\text{Front} = \text{Rear} = -1$ )
  - When the queue is fully empty and `dequeue()` operation is called queue underflow occurs.
- 
- **Queue Empty:**
  - Before deleting any element from queue check whether there is an element in the queue. If no element is present inside a queue & front & rear is set to -1 then queue is said to be empty.
  - **Size of queue = 4**
  - $\text{Front} = \text{Rear} = -1$



# Algorithm to insert element (enqueue Operation)

```
Step 1: IF REAR = MAX-1
        Write OVERFLOW
        Goto step 4
    [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
        SET FRONT = REAR = 0
    ELSE
        SET REAR = REAR + 1
    [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

# Algorithm to insert element (enqueue Operation)

- In Step 1, we first check for the *overflow* condition
- In Step 2, we check if the queue is empty
  - In case the queue is empty, then both **FRONT** and **REAR** are set to zero so that the new value can be stored at the 0<sup>th</sup> location
  - Otherwise, if the queue already has some values, then **REAR** is incremented so that it points to the next location in the array
- In Step 3, the value is stored in the queue at the location pointed by **REAR**

# Algorithm to delete element (dequeue Operation)

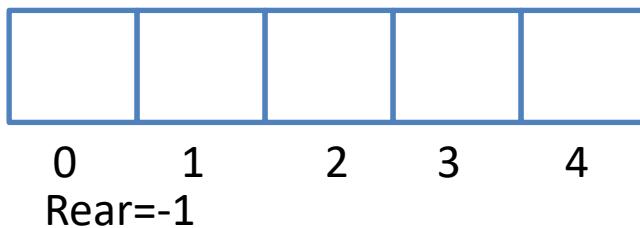
```
Step 1: IF FRONT = -1 OR FRONT > REAR
        Write UNDERFLOW
    ELSE
        SET VAL = QUEUE[FRONT]
        SET FRONT = FRONT + 1
    [END OF IF]
Step 2: EXIT
```

- In Step 1, we check for underflow condition. An underflow occurs if  $\text{FRONT} = -1$  or  $\text{FRONT} > \text{REAR}$
- However, if queue has some values, then  $\text{FRONT}$  is incremented so that it now points to the next value in the queue

Front=-1

Queue is empty

1

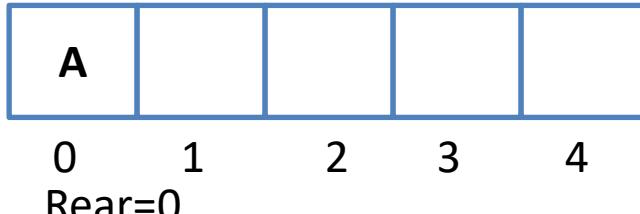


Rear=-1

Front=0

Insert A

2

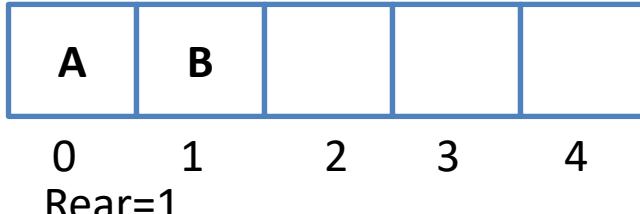


Rear=0

Front=0

Insert B

3

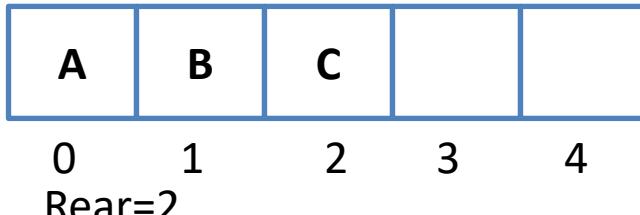


Rear=1

Front=0

Insert C

4

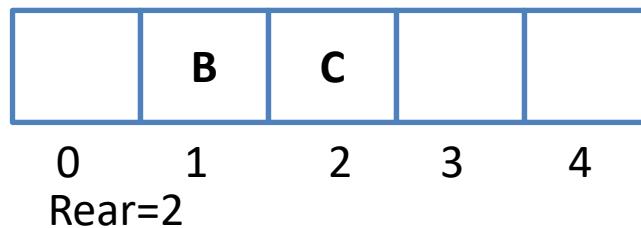


Rear=2

Front=1

Delete

5

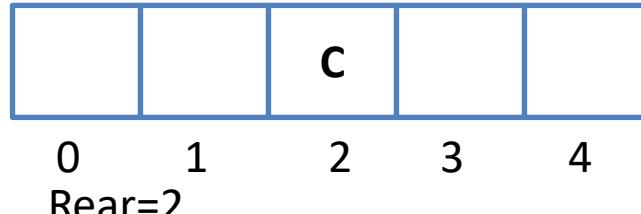


Rear=2

Front=2

Delete

6



Rear=2

Front=3

Delete

7



Rear=2

Queue is empty

**Entry point is called Rear &  
Exit point is called Front**

# Disadvantages of linear queue

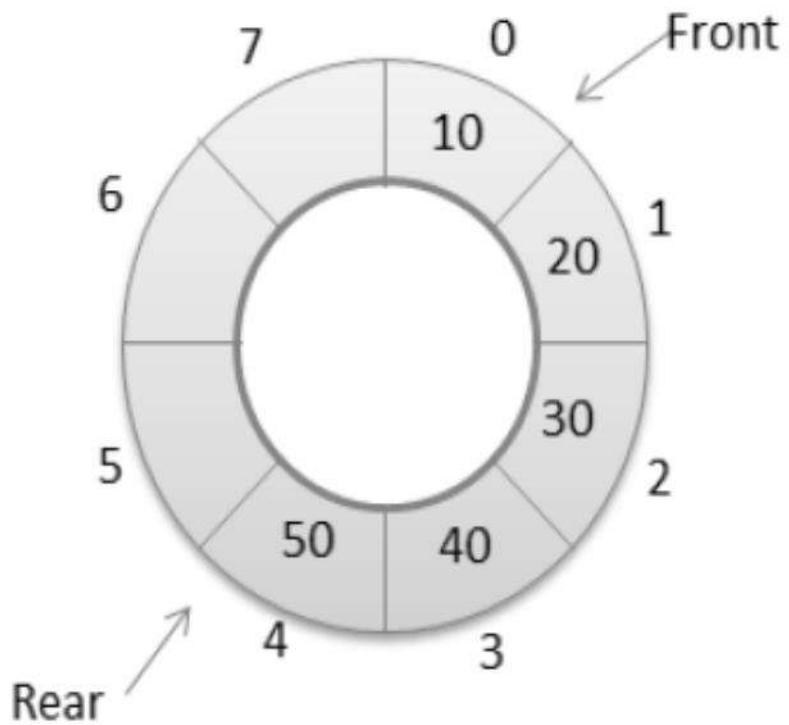
- On deletion of an element from existing queue, front pointer is shifted to next position.
- This results into virtual deletion of an element.
- By doing so memory space which was occupied by deleted element is wasted and hence inefficient memory utilization is occur.

## Overcome disadvantage of linear queue:

- To overcome the disadvantage of linear queue, the **circular queue** is used
- We can solve this problem by joining the front and rear end of a queue to make the queue as a circular queue
- A circular queue is a linear data structure. It follows the FIFO principle
- In a circular queue, the last node is connected back to the first node to make a circle

## Overcome disadvantage of linear queue:

- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in  $O(1)$  time.



# Representation Of Queues

- 1.Using an array
- 2.Using linked list

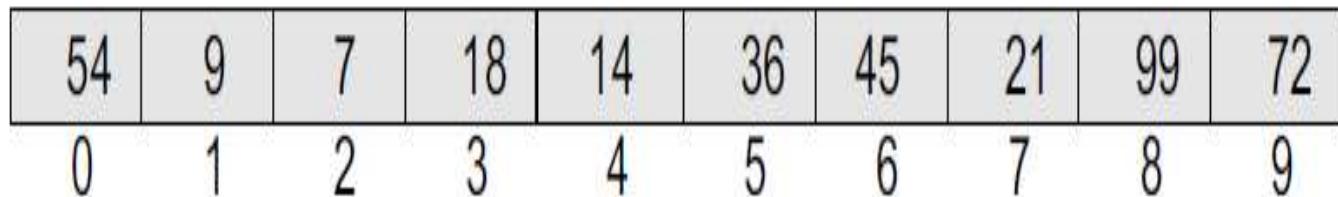


# Types Of Queue

1. Circular Queue
2. Dequeue (Double Ended Queue)

# CIRCULAR QUEUE

**FRONT = 0 and REAR = 9**



- If you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted

# CIRCULAR QUEUE

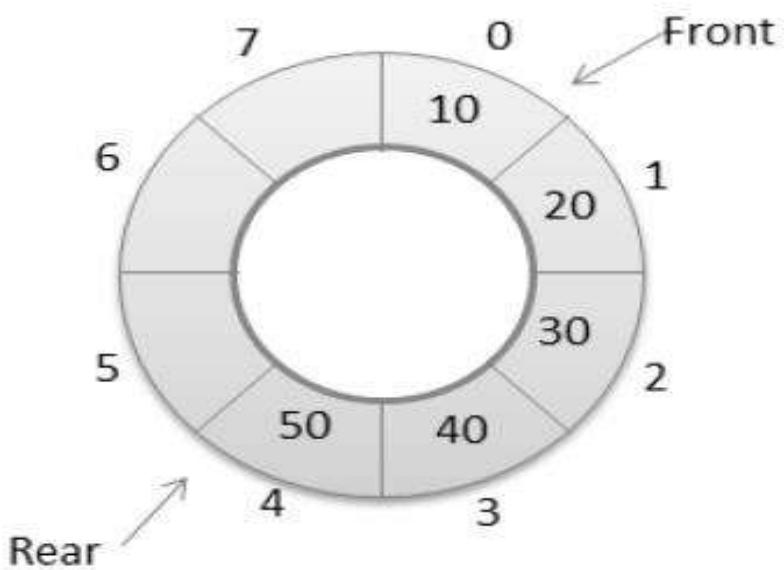
Front = 2 and REAR = 9

		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

- Consider a scenario in which two successive deletions are made
- Suppose we want to insert a new element in the queue
- Even though there is space available, the overflow condition still exists because the condition  $\text{rear} = \text{MAX} - 1$  still holds true
- This is a major drawback of a linear queue

# CIRCULAR QUEUE

- To resolve this problem, we have two solutions
- First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently
- But this can be very time-consuming, especially when the queue is quite large
- The second option is to use a circular queue
- In the circular queue, the first index comes right after the last index



# CIRCULAR QUEUE

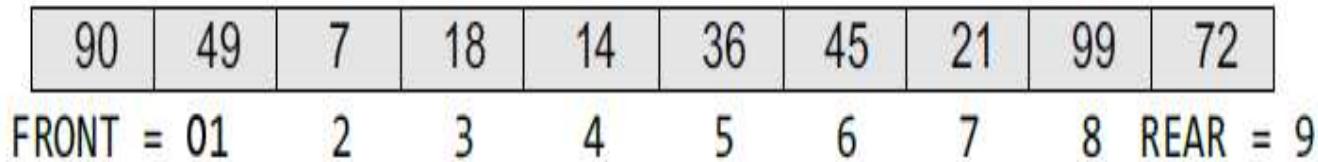
- A queue is called a circular queue in which the last node is connected back to the first node to form a cycle
- Circular queues are queues implemented in circular form rather than straight lines
- Circular queues overcome the problem of unutilized space in linear queues implemented as an array
- The main disadvantage of linear queue using an array is that when elements are deleted from the queue, new elements cannot be added in their place in the queue, i.e. the position cannot be reused

## CIRCULAR QUEUE IMPLEMENTATION

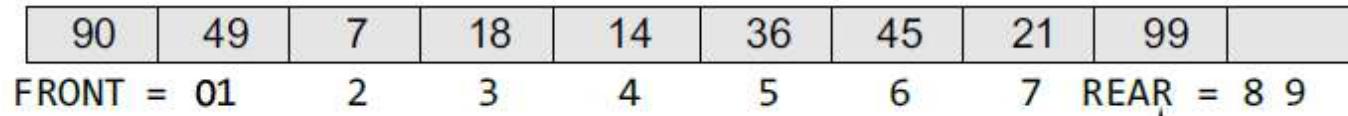
- After the rear reaches the last position, i.e. MAX-1 in order to reuse the vacant positions, we can bring rear back to the 0th position, if it is empty, and continue incrementing the rear in the same manner as earlier
- Thus rear will have to be incremented circularly
- For deletion, the front will also have to be incremented circularly

## Enqueue(Insert) operation on Circular Queue:

- A circular queue is implemented in the same manner as a linear queue is implemented
- For insertion, we now have to check for the following three conditions:
  1. If  $\text{front} = 0$  and  $\text{rear} = \text{MAX} - 1$ , then the circular queue is full



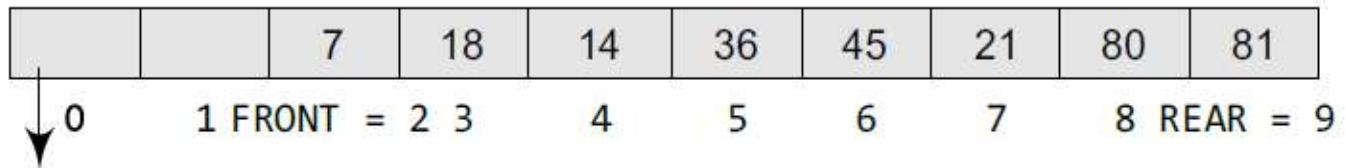
2. If  $\text{rear} \neq \text{MAX} - 1$ , then rear will be incremented and the value will be inserted



Increment rear so that it points to location 9 and insert the value here

## Enqueue(Insert) operation on Circular Queue:

3. If  $\text{front} \neq 0$  and  $\text{rear} = \text{MAX} - 1$ , then it means that the queue is not full. So, set  $\text{rear} = 0$  and insert the new element there



## Enqueue(Insert) operation on Circular Queue:

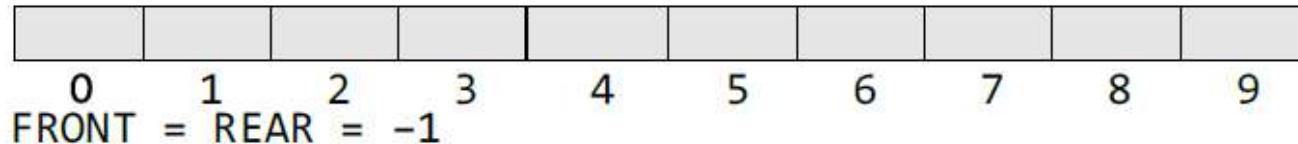
```
Step 1: IF FRONT = 0 and Rear = MAX - 1  
        Write "OVERFLOW"  
        Goto step 4  
    [End OF IF]  
Step 2: IF FRONT = -1 and REAR = -1  
        SET FRONT = REAR = 0  
    ELSE IF REAR = MAX - 1 and FRONT != 0  
        SET REAR = 0  
    ELSE  
        SET REAR = REAR + 1  
    [END OF IF]  
Step 3: SET QUEUE[REAR] = VAL  
Step 4: EXIT
```

## Enqueue(Insert) operation on Circular Queue:

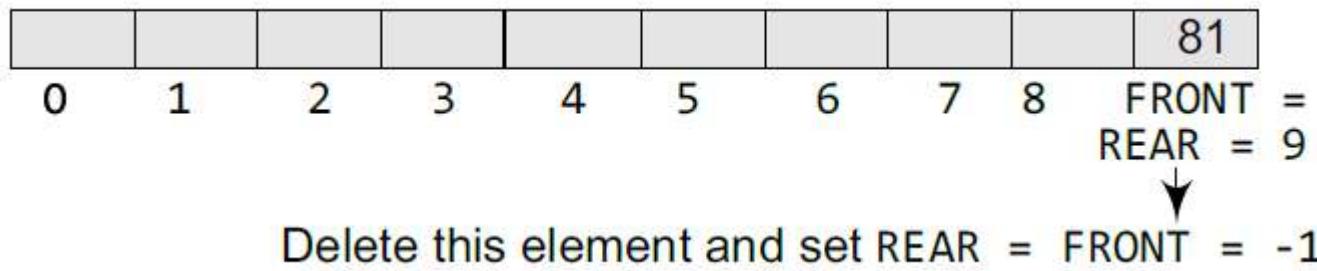
- In Step 1, we check for the overflow condition
- In Step 2, we make two checks.
  - ✓ First to see if the queue is empty
  - ✓ Second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end
- In Step 3, the value is stored in the queue at the location pointed by REAR

# Dequeue (Delete) operation on Circular Queue:

- If  $\text{front} = -1$ , then there are no elements in the queue. So, an underflow condition will be reported

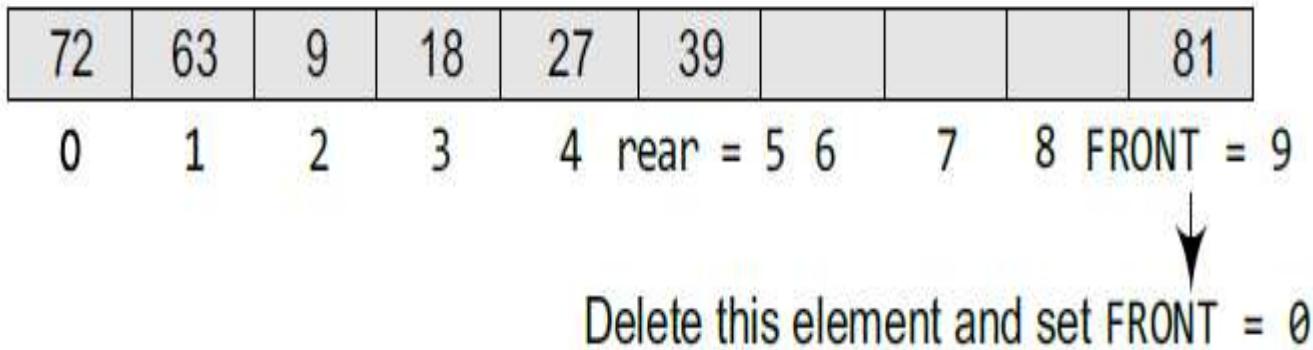


- If the queue is not empty and  $\text{front} = \text{rear}$ , then after deleting the element at the front the queue becomes empty and so front and rear are set to  $-1$



# Dequeue (Delete) operation on Circular Queue:

- If the queue is not empty and front = MAX-1, then after deleting the element at the front, front is set to 0



# Dequeue (Delete) operation on Circular Queue:

```
Step 1: IF FRONT = -1
        Write "UNDERFLOW"
        Goto Step 4
    [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
        SET FRONT = REAR = -1
    ELSE
        IF FRONT = MAX -1
            SET FRONT = 0
        ELSE
            SET FRONT = FRONT + 1
    [END of IF]
[END OF IF]
Step 4: EXIT
```

# Dequeue (Delete) operation on Circular Queue:

- In Step 1, we check for the underflow condition
- In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL
- In Step 3, we make two checks.
  1. First to see if the queue has become empty after deletion
  2. Second to see if FRONT has reached the maximum capacity of the queue
- The value of FRONT is then updated based on the outcome of these checks

# Double Ended queue (deQue)

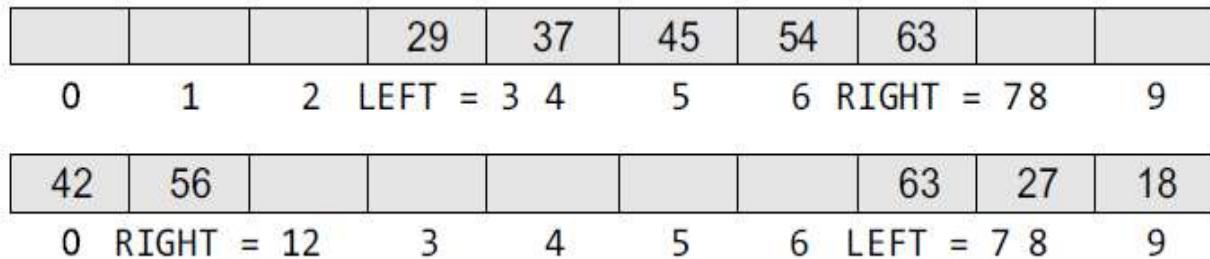
- A deque (pronounced as ‘deck’ or ‘dequeue’) is a list in which the elements can be inserted or deleted at either end
- It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end
- However, no element can be added and deleted from the middle

# Double Ended queue (deQue)

- In the computer's memory, a deque is implemented using either a circular array or a circular doubly linked list
- In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque
- The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N-1] is followed by Dequeue[0]

# Double Ended queue (deQue)

- There are two variants of a double-ended queue



- **Input restricted deque-** In this deQueue, insertions can be done only at one of the ends, while deletions can be done from both ends
- **Output restricted deque-** In this deQueue, deletions can be done only at one of the ends, while insertions can be done on both ends

## Applications of Queue

- Serving requests on a single shared resource, like a printer, CPU task scheduling, etc
- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free



# **BCS202P- Data Structures and Algorithms Lab**

**Dr. Priyanka N**

**Assistant Professor Senior Grade I**

**School of Computer Science & Engineering**

**VIT, Vellore.**

# **BCS202P- Data Structures and Algorithms Lab**

## **COURSE OBJECTIVE**

1. To impart basic concepts of data structures and algorithms.
2. To differentiate linear, non-linear data structures and their operations.
3. To comprehend the necessity of time complexity in algorithms.



# **BCS202P- Data Structures and Algorithms Lab**

## **COURSE OUTCOMES**

1. Apply appropriate data structures to find solutions to practical problems.
2. Identify suitable algorithms for solving the given problems.

# **BCS202P- Data Structures and Algorithms Lab - Experiments**

1.	Implementation of stack data structure and its applications
2.	Implementation of queue data structure and its applications
3.	Implementation linked list and its application
4.	Implementation of searching algorithms
5.	Implementation of sorting algorithms
6.	Binary Tree Traversal implementation
7.	Binary Search Tree implementation
8.	Graph Traversal – Depth First Search and Breadth First Search algorithm
9.	Minimum Spanning Tree – Prim's and Kruskal's algorithm
10.	Single Source Shortest Path Algorithm - Dijkstra's algorithm
Total Laboratory Hours	
30 hours	

# **BCS202L- Data Structures and Algorithms**

## **Text Books:**

- I.** Mark A. Weiss, Data Structures & Algorithm Analysis in C++, 4 th Edition, 2013,Pearson Education.

## **Reference Books:**

- I.** Alfred V. Aho, Jeffrey D. Ullman and John E. Hopcroft, Data Structures and Algorithms,1983, Pearson Education.
- 2.** Horowitz, Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, 2008, 2<sup>nd</sup> Edition, Universities Press.
- 3.** Thomas H. Cormen, C.E. Leiserson, R L. Rivest and C. Stein, Introduction to Algorithms, 2009, 3<sup>rd</sup> Edition, MIT Press.

# Basic Terminologies in DS

**Data:** Data are simply values or sets of values.

**Data items:** Data items refers to a single unit of values.

**Group:** Data items that are divided into sub-items are called **Group** items.

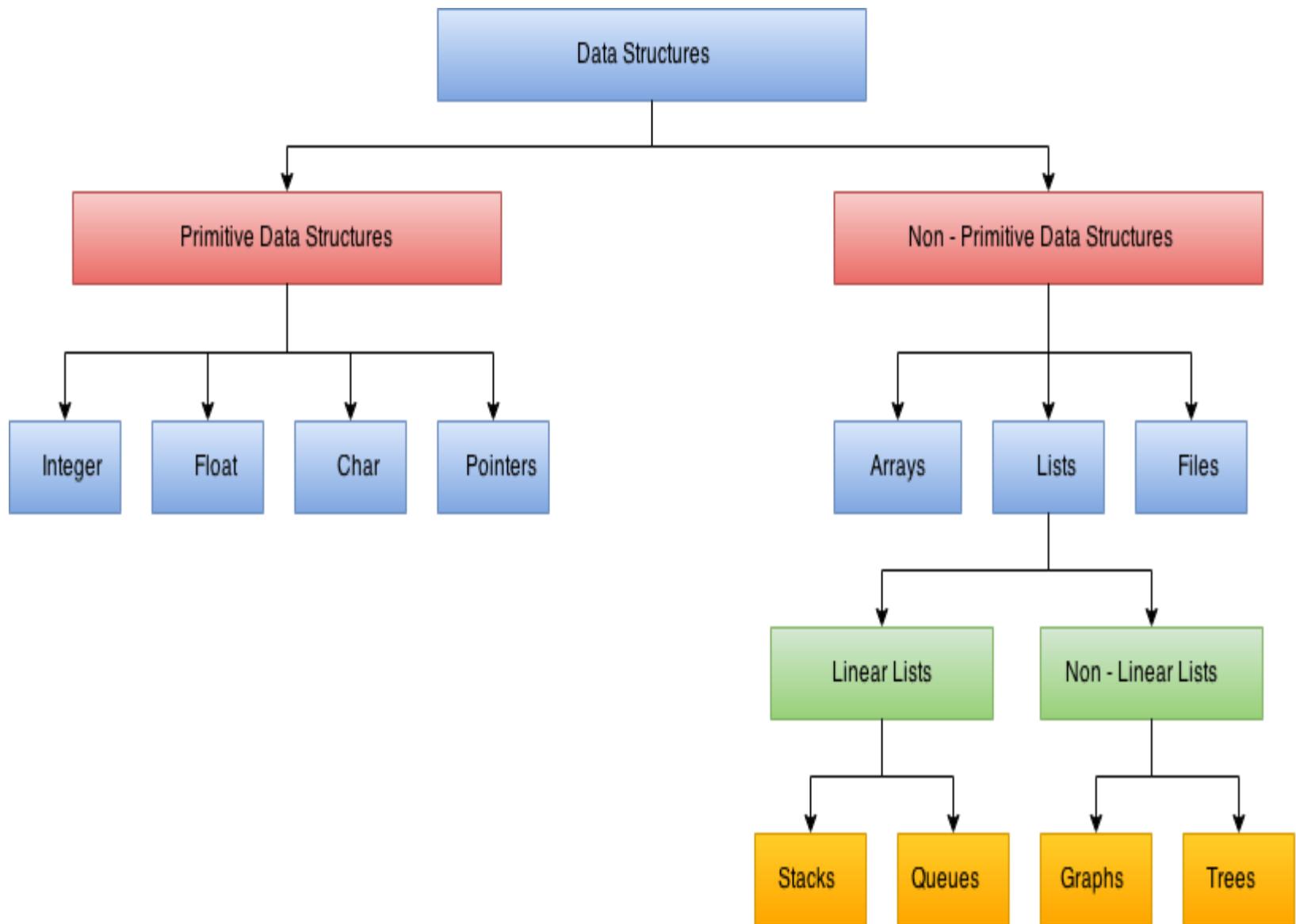
**Elementary items:** Data items that are not able to divide into sub-items are called Elementary items.

**Entity:** An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric.

**Field** is a single elementary unit of information representing an attribute of an entity.

**Record** is the collection of field values of a given entity.

**File** is the collection of records of the entities in a given entity set.

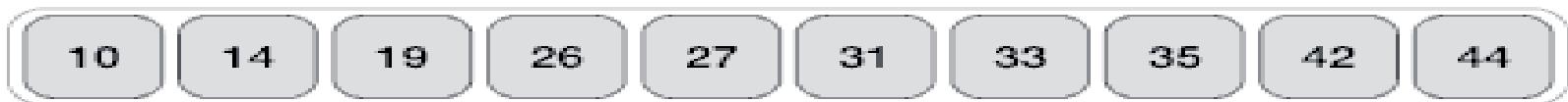


# Implementation of Searching Algorithm

- Linear Search

- Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Linear Search



## **Linear search( array[], search element)**

```
{  
    for i=1 to n do  
    {  
        if( array[i]==search element)  
        {  
            return i;  
        }  
    }  
    return 0;  
}
```

# Binary Search

- Binary search is a fast search algorithm works on the principle of **divide and conquer**,
- Data should be compulsory in the sorted form

# How Binary Search Works?

- Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned.
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

## Binary search( array[], x, Low, high, mid)

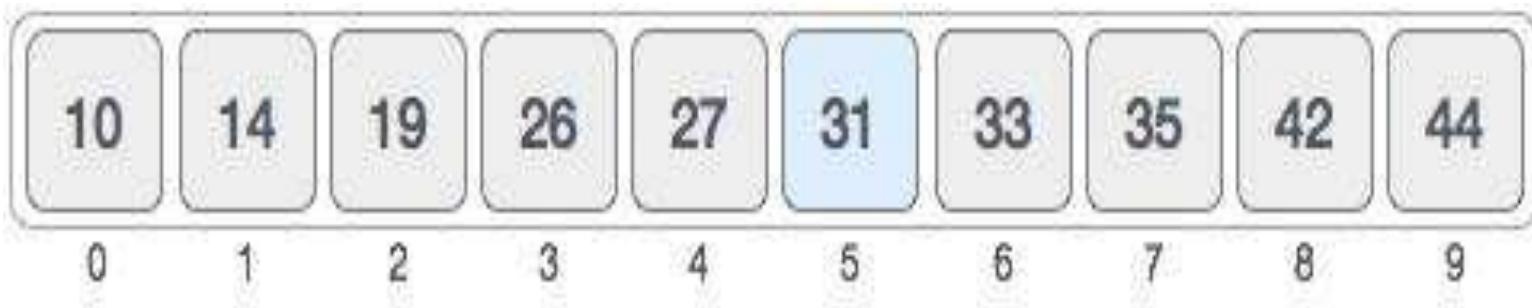
```
{  
    Low=1, high=n;  
    While(low<=high)  
    {  
        mid=(low+high)/2;// Finding mid value  
        if(x==a[mid])  
            return mid;  
        else if(x<a[mid])  
            high=mid-1;  
        else  
            low=mid+1;  
    }  
    return 0;  
}
```

# How Binary Search Works?

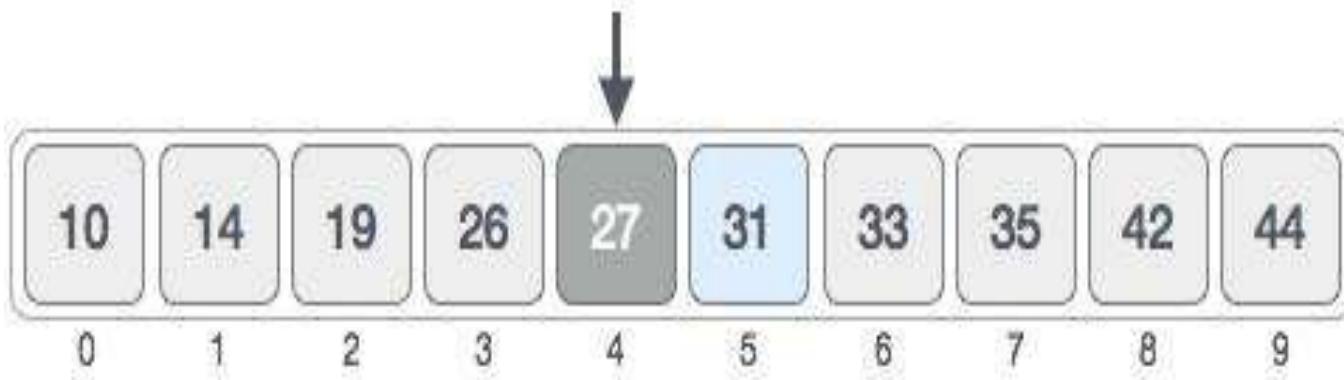
For a binary search to work, it is mandatory for the target array to be sorted.

## Example:

- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



# Binary Search



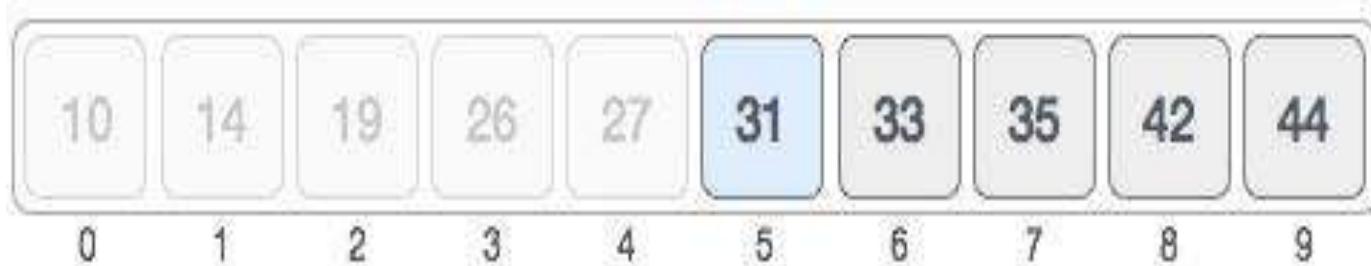
**First, we shall determine half of the array by using this formula –**

$$\text{mid} = (\text{low} + \text{high}) / 2$$

**Here it is,  $(0 + 9) / 2 = 4$  (integer value of 4.5).  
So, 4 is the mid of the array.**

# Binary Search

- Now we compare the value stored at location 4, with the value being searched, i.e. 31.
- We find that the value at location 4 is 27, which is not a match.
- As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



# Binary Search

We change our low to mid + 1 and find the new mid value again.

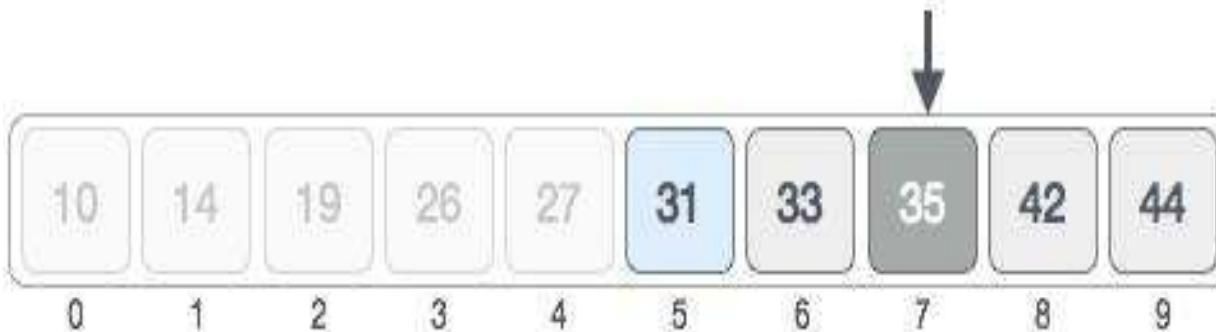
**low = mid + 1**

**High = remains same**

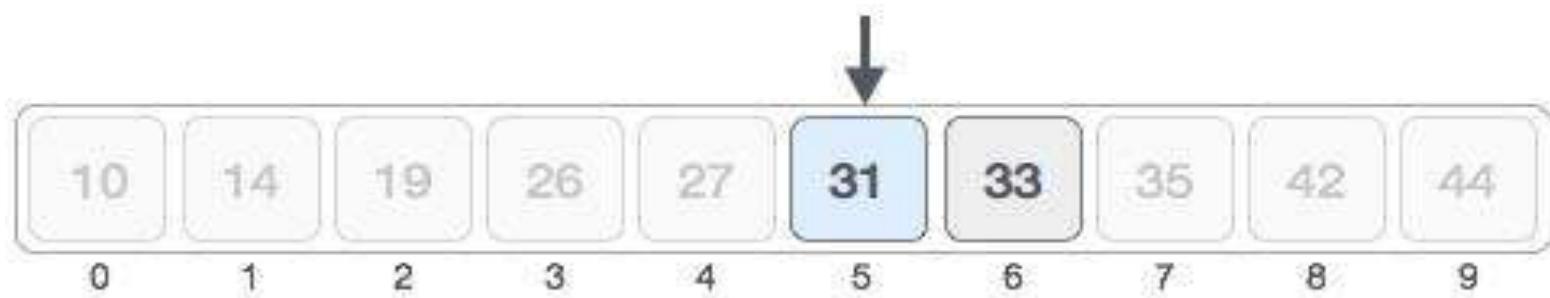
**mid = (low + high) / 2**

$$\text{Low} = 4 + 1 = 5; \text{mid} = (5 + 9) / 2 = 7$$

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

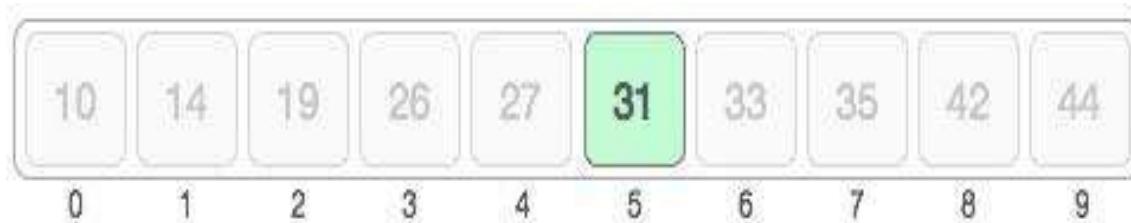


- The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.
- Hence, we calculate the mid again. This time it is 5.



# Binary Search

- We compare the value stored at location 5 with our target value. We find that it is a match.
- We conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.



# Visualization of Linear and Binary Search

<https://www.cs.usfca.edu/~galles/visualization/Search.html>

# Selection sort

- IDEA:
  - Find the smallest element in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.
- Selection sort is:
  - One of the simplest sorting techniques.
  - Good algorithm to sort a small number of elements

## Selection Sort



# Algorithm Selection Sort (A [], n)

```
{  
    For i = 0, i<n-1,i++)  
    {  
        int min=i;  
        for(j=i+1,j<n,j++)  
        {  
            if(a[j]<a[min])  
            {  
                min=j;  
            }  
        }  
        if(min!=i)  
        {  
            swap(a[i],a[min])  
        }  
    } }
```

# Selection Sort - Complexity

Case	Time Complexity
<b>Best Case</b>	$O(n^2)$
<b>Average Case</b>	$O(n^2)$
<b>Worst Case</b>	$O(n^2)$



# **BCS202L & BCS202P- Data Structures and Algorithms**

**Dr. Priyanka N**

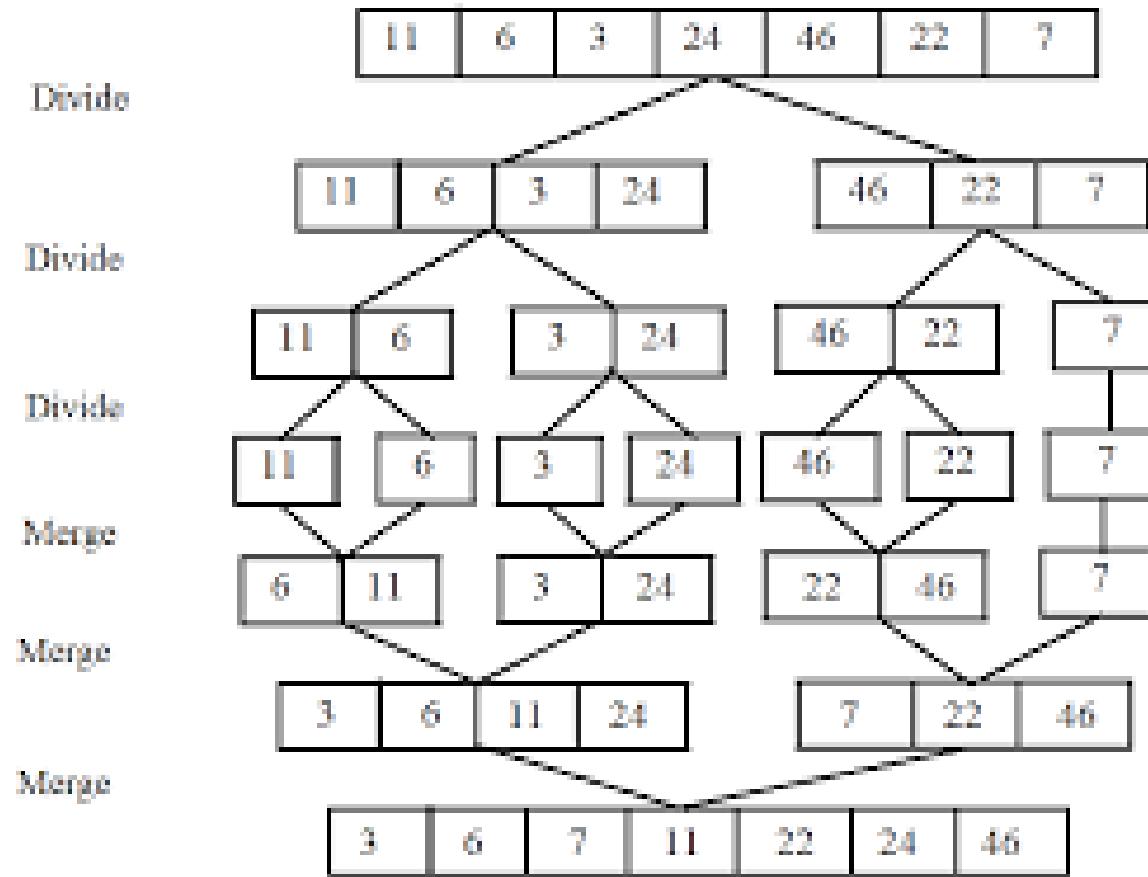
**Assistant Professor Senior Grade I**

**School of Computer Science & Engineering**

**VIT, Vellore.**

# MERGE SORT

- Works under Divide and Conquer strategy
- **IDEA:** Given a sequence of  $n$  elements  $a[1], a[2], \dots, a[n]$ , the general idea is to imagine them split into two sets  $a[1], a[2], \dots, a[n/2]$  and  $a[n/2+1], \dots, a[n]$ .
- Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of  $n$  elements



```
MergeSort(A,lb,ub)
{
    if(lb<ub)
    {
        mid=(lb+ub)/2;
        MergeSort(A,lb,mid);
        MergeSort(A, mid+ 1,ub)
        MergeSort(A,lb, mid,ub)
    }
}
```

```

MergeSort(A,lb,mid,ub)
{
    i=lb;
    J=mid+1;
    k=lb;
    while(i<=mid && j<=ub)
    {
        if(a[i]<=a[j])
        {
            b[k]=a[i];
            i++;
        }
        else
        {
            b[k]=a[j];
            j++;
        }
        k++;
    }
}

if(i>mid)
{
    while(j<=ub)
    {
        b[k]=a[j];
        j++;
        k++;
    }
}
else
{
    while(i<=mid)
    {
        b[k]=a[i];
        i++;
        k++;
    }
}
for(k=lb;k<=ub;k++)
{
    a[k]=b[k];
}
}

```

# Quick Sort

- Quick Sort is a famous sorting algorithm
- It sorts the given data items in ascending order
- It uses the idea of divide and conquer approach
- It follows a recursive algorithm
- a = Linear Array in memory
- beg = Lower bound of the sub array in question
- end = Upper bound of the sub array in question

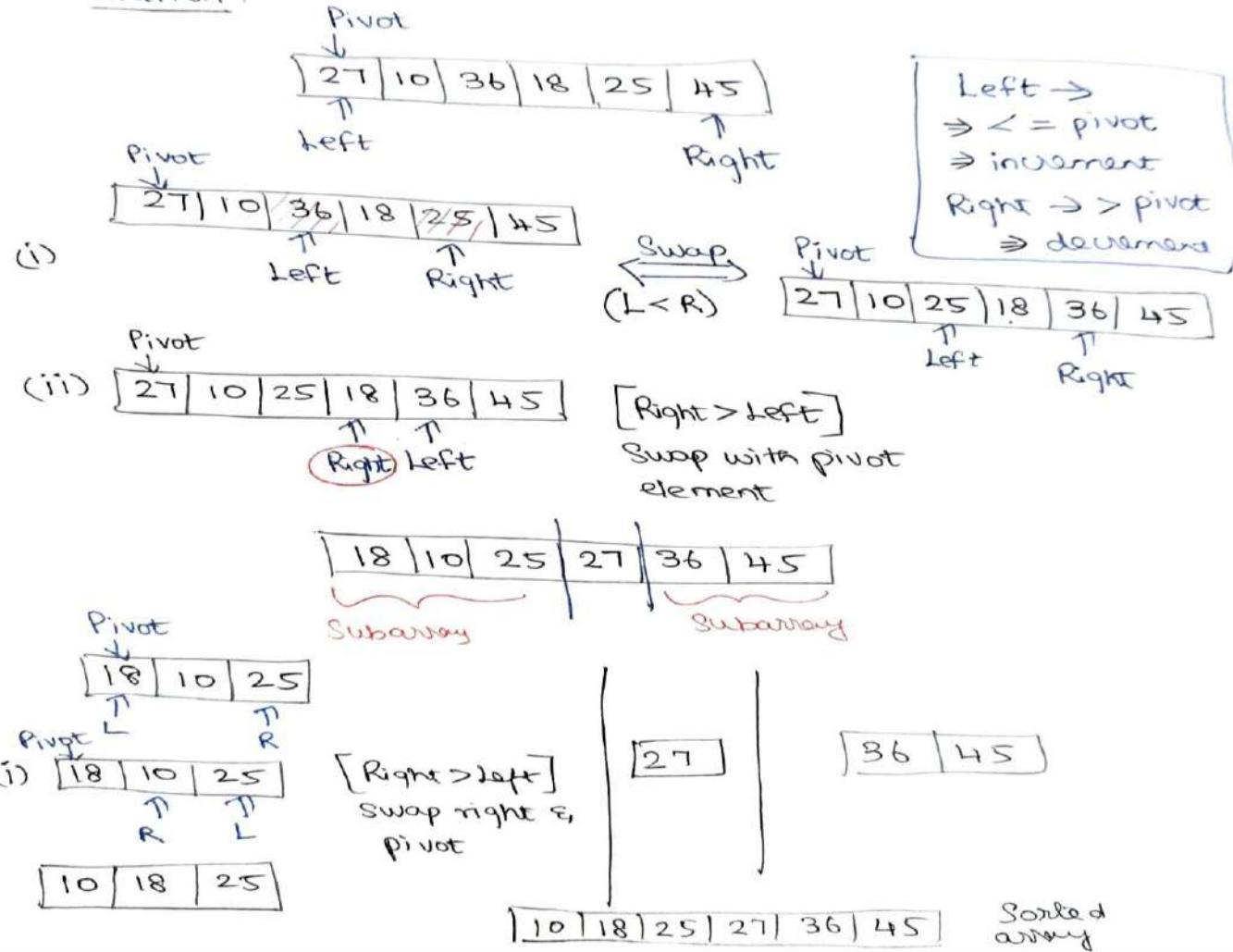
# How Does Quick Sort Works?

- It divides the given array into two sections using a partitioning element called as pivot
- The division performed is such that-
  - ❑ All the elements to the left side of pivot are smaller than pivot
  - ❑ All the elements to the right side of pivot are greater than pivot
- After dividing the array into two sections, the pivot is set at its correct position
- Then, sub arrays are sorted separately by applying quick sort algorithm recursively

## Quick Sort

27	10	36	18	25	45
----	----	----	----	----	----

Solution:



```
partition(A,lb,ub)
{
    pivot = a[lb];
    start=lb;
    end=ub;
    while(start<end)
    {
        while(a[start]<=pivot)
        {
            start++;
        }
        while(a[end]>pivot)
        {
            end--;
        }
        if(start<end)
        {
            swap(a[start],a[end])
        }
    }
    swap(a[lb],a[end])
    Return end
}

Quicksort(A,lb,ub)
{
    If(lb<ub)
    {
        loc=partition(A,lb,ub)
        Quicksort(A,lb,loc-1)
        Quicksort(A,loc+1,ub)
    }
}
```

# Counting sort

- This sorting technique doesn't perform sorting by comparing elements
- It performs sorting by counting objects having distinct key values like hashing
- After that, it performs some arithmetic operations to calculate each object's index position in the output sequence

# EXAMPLE

2	9	7	4	1	8	4
---	---	---	---	---	---	---

1. Find the maximum element from the given array. Let **max** be the maximum element.

max	9	2	7	4	1	8	4
-----	---	---	---	---	---	---	---

2. Now, initialize array of length **max + 1** having all 0 elements. This array will be used to store the count of the elements in the given array.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Count array

3. Now, we have to store the count of each array element at their corresponding index in the count array.

The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2. Hence, 2 is stored at the 4<sup>th</sup> position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3<sup>rd</sup> position.

Given array	2	9	7	4	1	8	4
-------------	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8 9

Count array	0	1	1	0	2	0	0	1	1	1
-------------	---	---	---	---	---	---	---	---	---	---

Count of each stored element

Now, store the cumulative sum of **count** array elements. It will help to place the elements at the correct index of the sorted array.

0	1	2	3	4	5	6	7	8	9
0	1	2	0	2	0	0	1	1	1

$$1+1=2$$

0	1	2	3	4	5	6	7	8	9
0	1	2	2	2	0	0	1	1	1

$$2+0=2$$

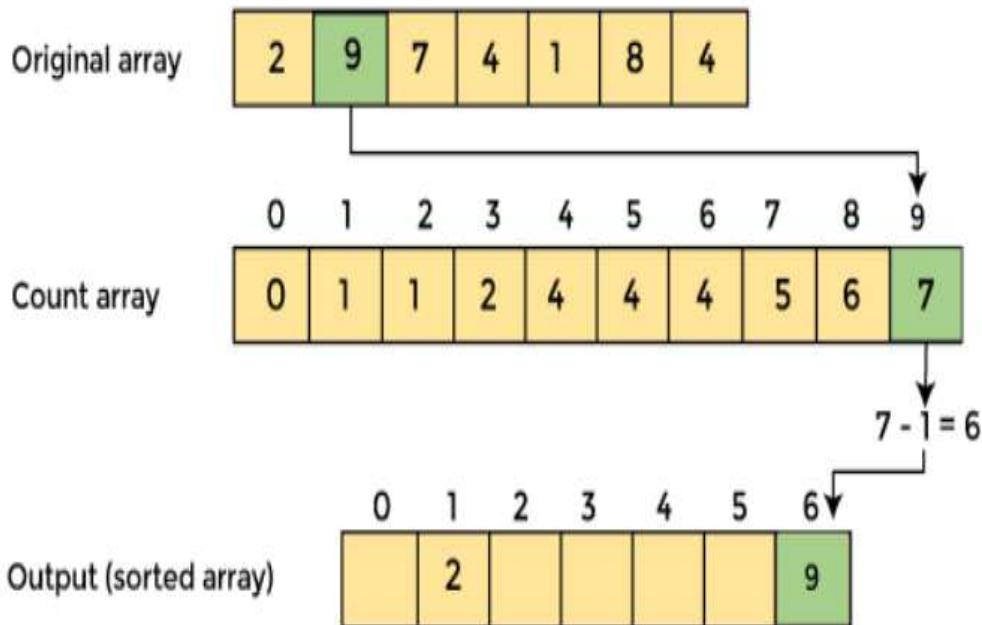
Similarly, the cumulative count of the count array is -

0	1	2	3	4	5	6	7	8	9
0	1	2	2	4	4	4	5	6	7

Cumulative count

4. Now, find the index of each element of the original array

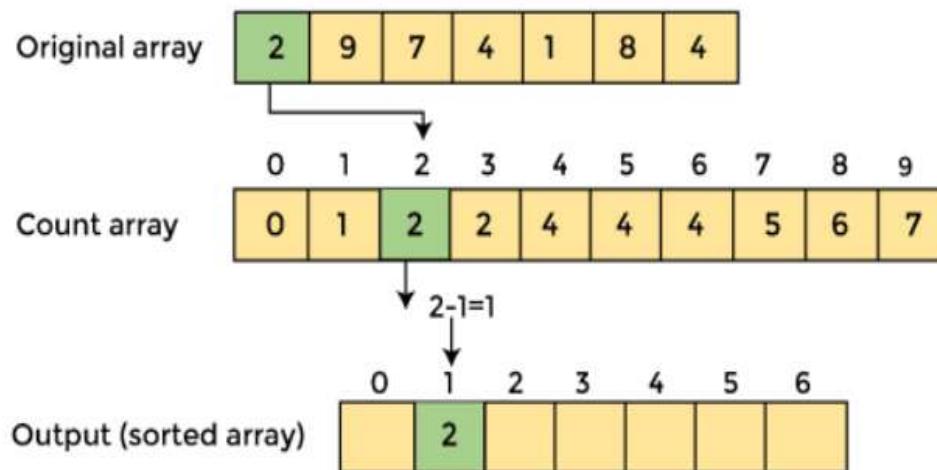
For element 9



After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.

After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.

### For element 2



Similarly, after sorting, the array elements are -

Output (sorted array)

0	1	2	3	4	5	6
1	2	4	4	7	8	9

Now, the array is completely sorted.

```
Void countingSort(int A[], int B[], int k, int n)
{
    int c[k+1]
    for (int i=0; i<=k; i++) //Initialize array to “0”
    {
        c[i]=0
    }
    for (int i=0; i<n; i++) //Calculate count value of the array
    {
        c[A[i]]= c[A[i]]+1
    }
    for (int i=1; i<=k; i++) //Calculate cumulative count value of the array
    {
        c[i]= c[i-1]+c[i]
    }
    for (int i=n-1; i>=0; i--)
    {
        B[--c[A[i]]]= A[i]
    }
    for (int i=0; i<n; i++)
    {
        A[i]= B[i]
    }
}
```

# Counting sort

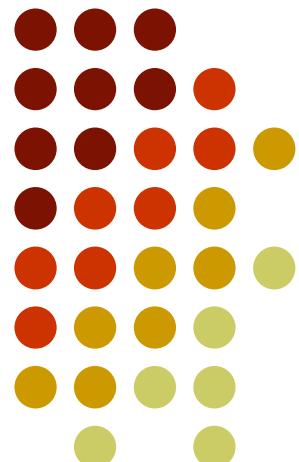
Case	Time
Best Case	$O(n + k)$
Average Case	$O(n + k)$
Worst Case	$O(n + k)$

# Data Structures and Algorithms (BCSE202L)

*Module 2: List*

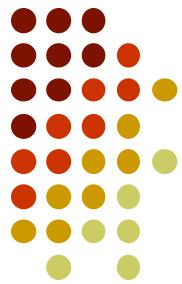
---

**Dr. Priyanka N**



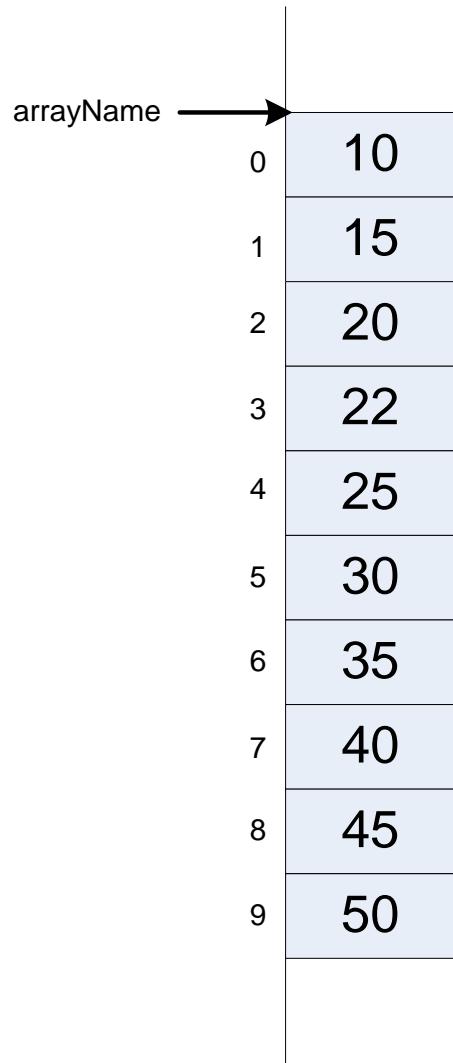
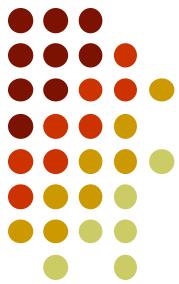


- Introduction to linked list
- Array versus linked list
- Linked lists in C
- Types of linked lists
  - Single linked list
  - Doubly linked list
  - Circular linked list



# Arrays versus Linked Lists

# Array: Contagious Storage

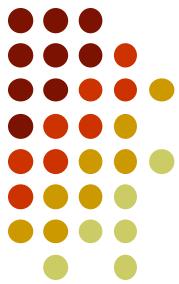


# Array versus Linked Lists



- In arrays
  - elements are stored in a contiguous memory locations
  - Arrays are static data structure unless we use dynamic memory allocation
  - Arrays are suitable for
    - Inserting/deleting an element at the end.
    - Randomly accessing any element.
    - Searching the list for a particular value.

# Array versus Linked Lists

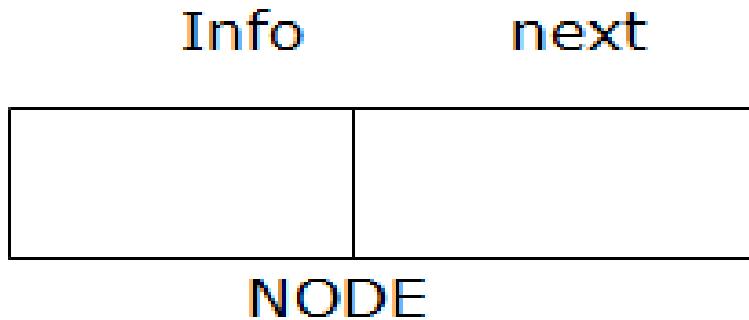


- **In Linked lists**
  - adjacency between any two elements are maintained by means of links or pointers
- It is essentially a dynamic data structure
- Linked lists are suitable for
  - Inserting an element at any position.
  - Deleting an element from any where.
  - Applications where sequential access is required.
  - In situations, where the number of elements cannot be predicted beforehand.

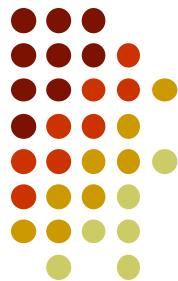


# Linked list

- A linked list is a linear data structure where data is stored in the form of a node



- The info field represents the data and the next field represents the address of the next node



# Types of Linked List

- Singly Linked list
- Doubly Linked List
- Circular Linked list



# SINGLY LINKED LIST



# Linked list

## Representation of Linked List in Memory

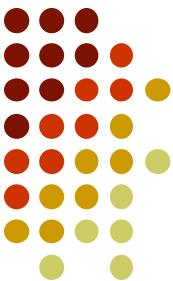
There are two ways to represent a linked list in memory

- Static representation using an array
- Dynamic representation



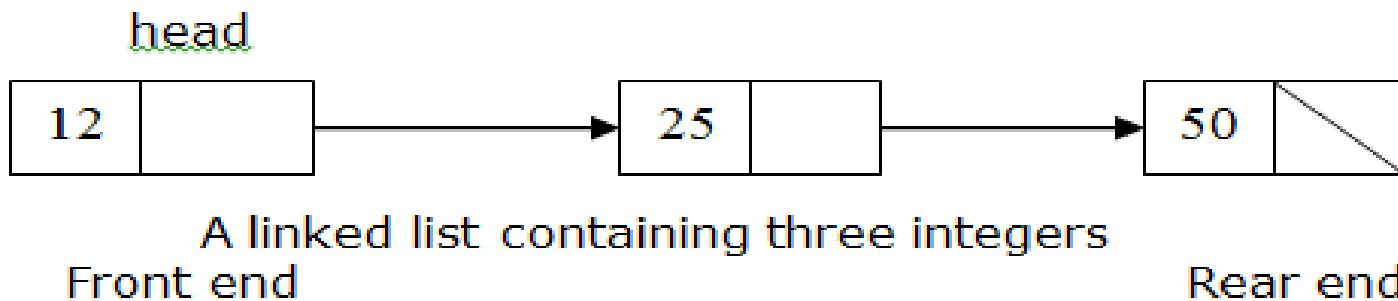
# Static Representation

- Static representation of a linked list maintains two arrays:
  - one array for the data
  - other for next



# Dynamic Representation

- The efficient way of representing a linked list is using the pointer, one such simple representation is shown below:

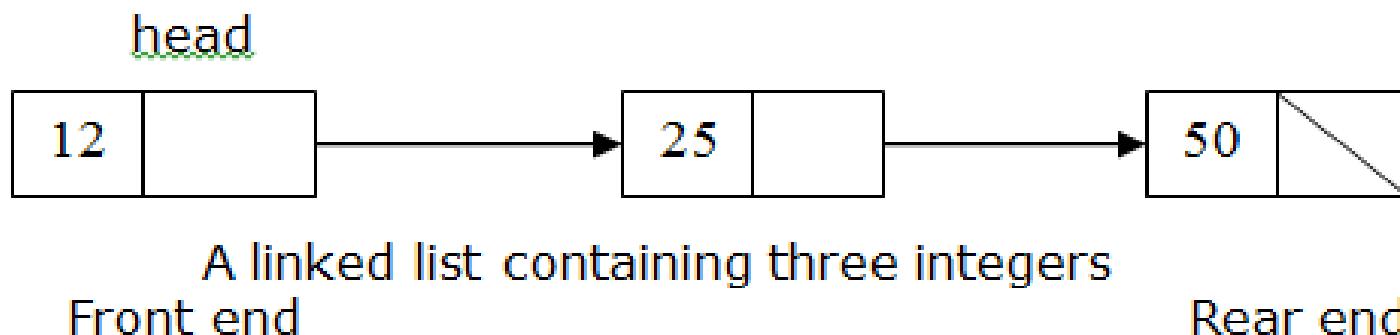


- Here, the head is a pointer that points to the head node in the list and the last node's next part contains a NULL denoting that it is the end of the list

# Basic Condition



- Head is a pointer that points to the first node in the list
  - If the list is empty head=NULL
  - The last node's next field contains NULL indicating that it's the last node in the list

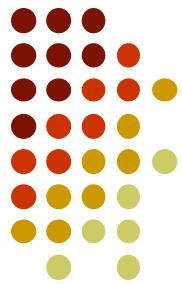


Note: Whatever operations we do on the linked list  
we should not violate the above basic conditions



# Operations on Singly Linked list

- Creation of the list
- Insertion at the Front end
- Insertion at the Rear end
- Insertion at a specified position
- Deletion at the Front end
- Deletion at the Rear end
- Deletion of a specified node
- Traversal ( Display )



# Creation of list

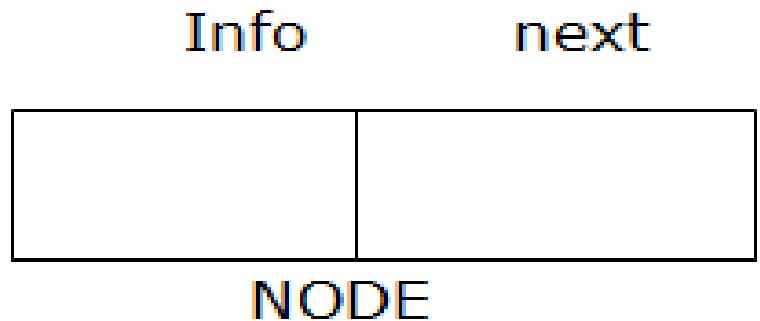
struct slist

{

    int info;

    struct slist \*next;

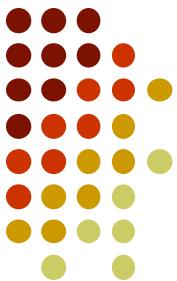
};



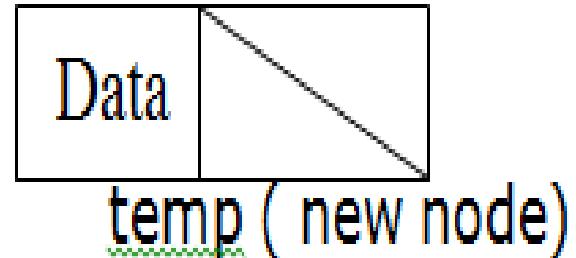


# INSERTION AT FRONT END

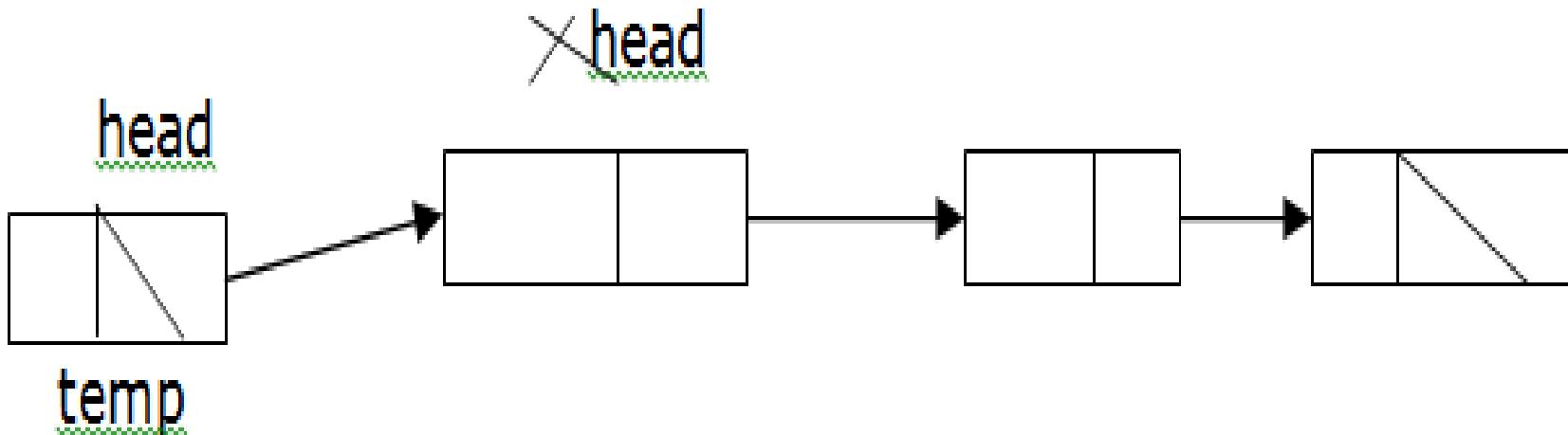
# Insertion at the Front end

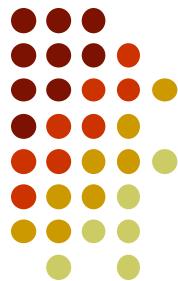


Case 1: When the list is empty  
i.e head=NULL



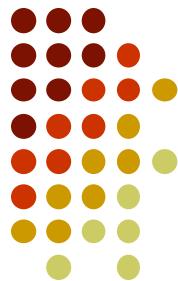
Case 2: When the list is not empty





# Insertion at the Front end

1. Create a new node ( say temp) containing the information to be inserted
2. `slist newnode;`
3. `newnode->info=data;`
4. `newnode->next=NULL;`
5. If the list is empty i.e. `head = NULL` then make the new node as the head node in the list
6. `head=newnode;`



# Insertion at the Front end

7. If the list is not empty then,

- Store the address of the head node in the next part of the new node and
- Make the new node as the head node

```
head = newnode;
```

```
newnode->next=head;
```

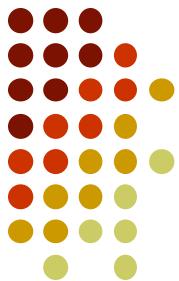
# Insertion at the Front end



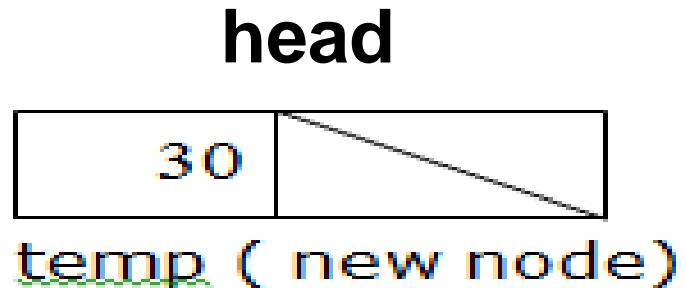
1. create a new node ( say temp) containing the information to be inserted.
2. node temp;
3. temp->info=data;
4. temp->next=NULL;



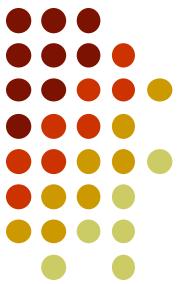
# Insertion at the Front end



5. If the list is empty i.e. head = NULL then make the new node as the head node in the list
6. head=temp;

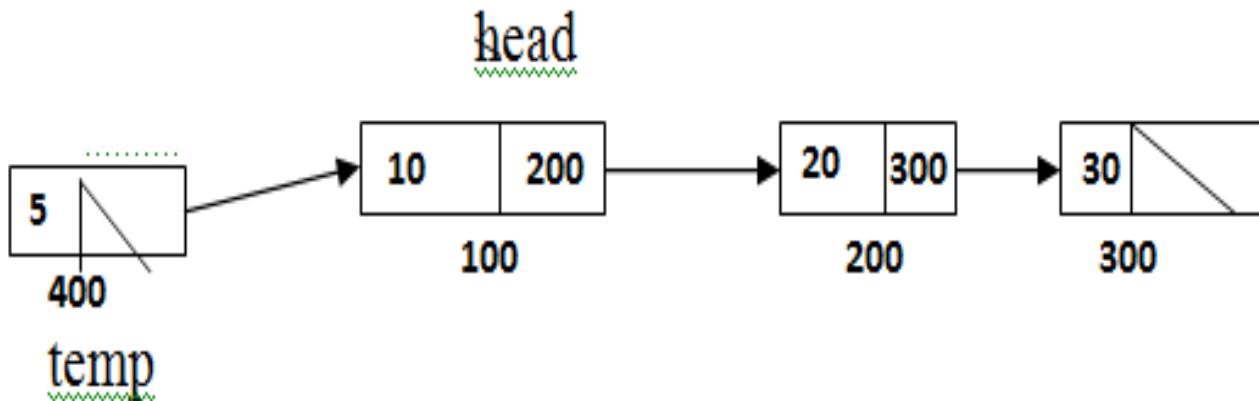


# Insertion at the Front end

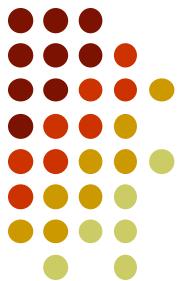


7. If the list is not empty then
  - Store the address of head node in the next part of the new node

`temp->next=head`

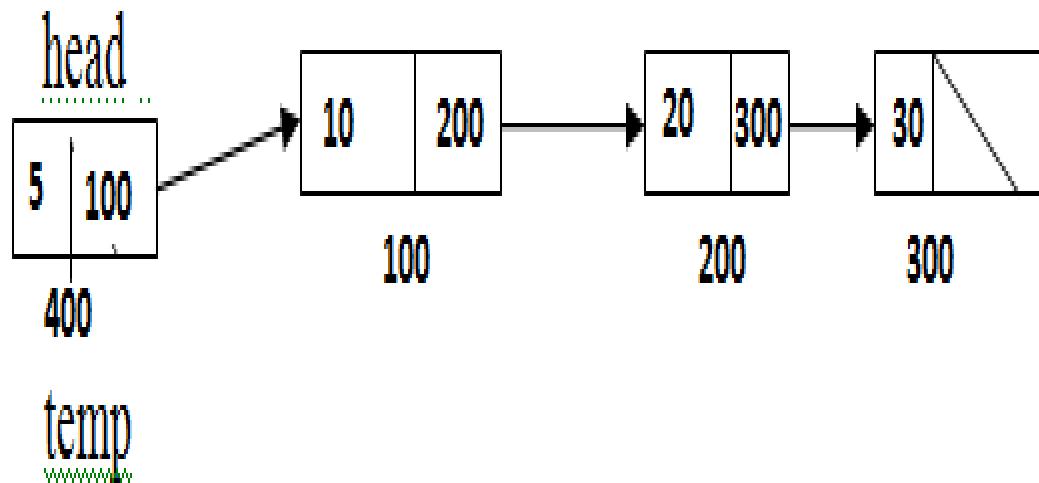


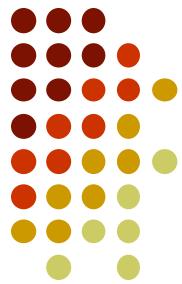
# Insertion at the Front end



8. Make the new node as the head node

```
head = temp;
```



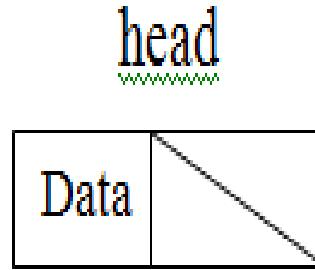


# INSERTION AT REAR END

# Insertion at Rear end

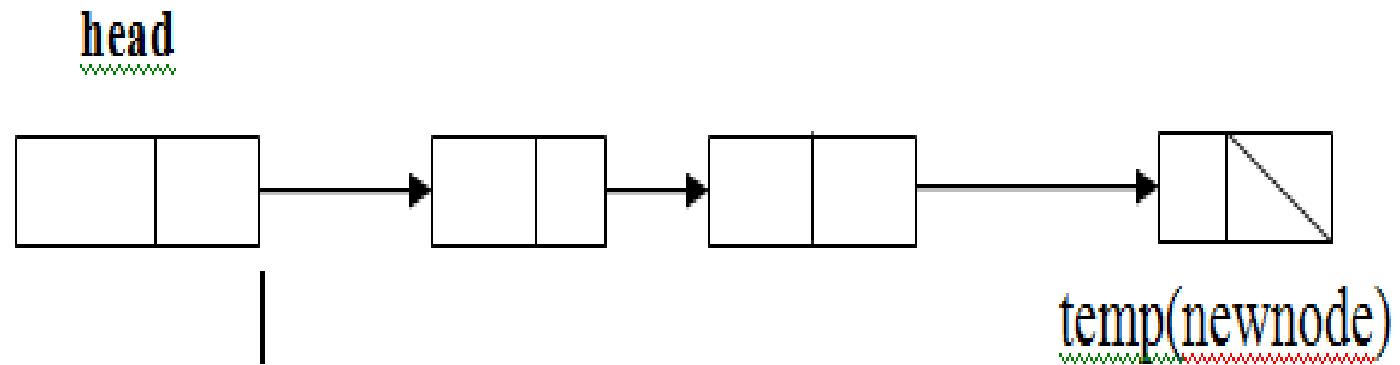


Case 1: When the list is empty



temp (new node)

Case 2: When the list is not empty



# Insertion at the Rear end



1. create a new node ( say temp) containing the information to be inserted.
2. node temp;
3. temp->info=data;
4. temp->next=NULL;
5. If the list is empty i.e. head = NULL then make the new node as the head node in the list
6. head=temp;

# Insertion at the Rear end



7. If the list is not empty then,

- Create a temporary pointer (say curn), store the head node's address, and move to the end of the list
- After reaching the last node store the address of the new node in the next part of the last node

slist curn;

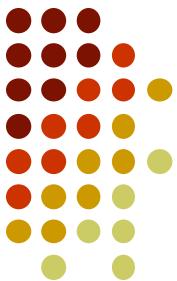
curn=head;

While (curn->next!=NULL)

    curn=curn->next;

    curn->next=newnode;

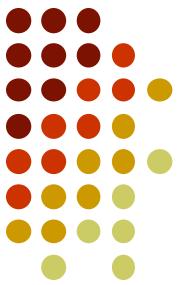
# Insertion at the Rear end



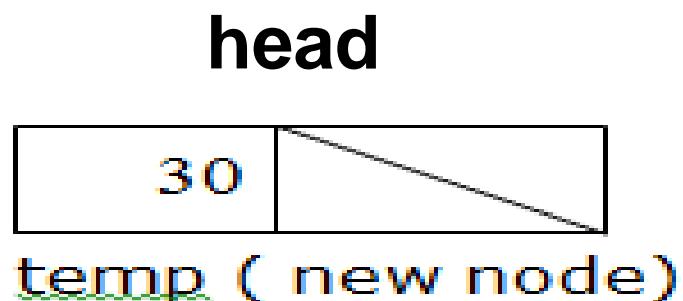
1. Create a new node ( say temp) containing the information to be inserted
2. node temp;
3. temp->info=data;
4. temp->next=NULL;



# Insertion at the Rear end



5. If the list is empty i.e. head = NULL then make the new node as the head node in the list
6. head=temp;



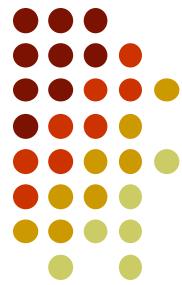
# Insertion at the Rear end



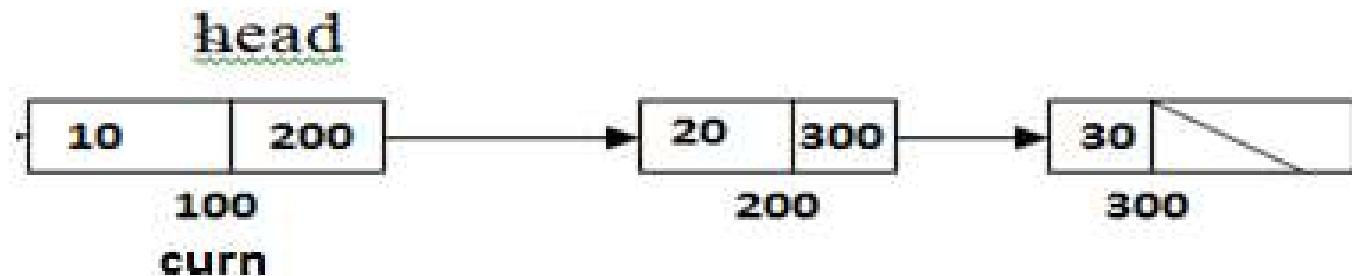
7. If the list is not empty then,

- Create a temporary pointer (say curn), store the head node's address, and move it to the end of the list
- After reaching the last node store the address of the new node in the next part of the last node

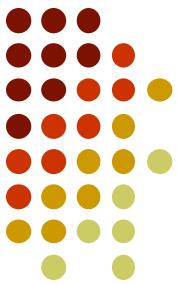
# Insertion at the Rear end



- slist curr =head;

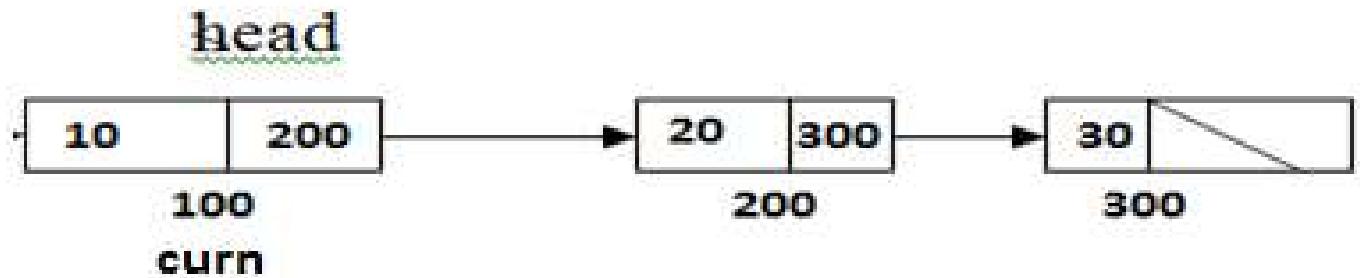


# Insertion at the Rear end



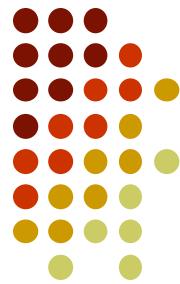
```
While (curn->next!=NULL)
```

```
curn=curn->next;
```



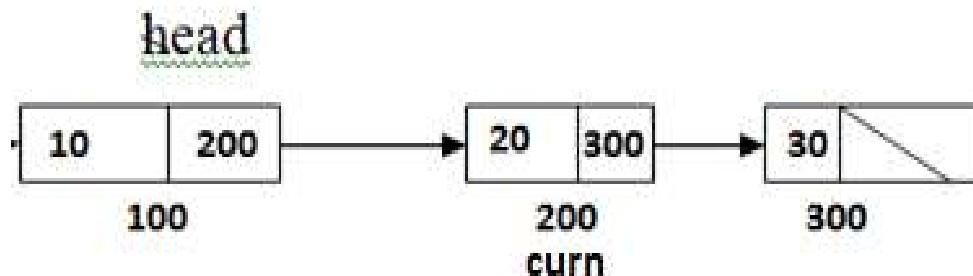
curn-> next is 200 which is not equal to NULL so curn moves to next node

# Insertion at the Rear end



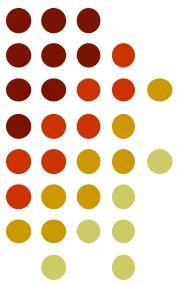
While (curn->next!=NULL)

curn=curn->next;



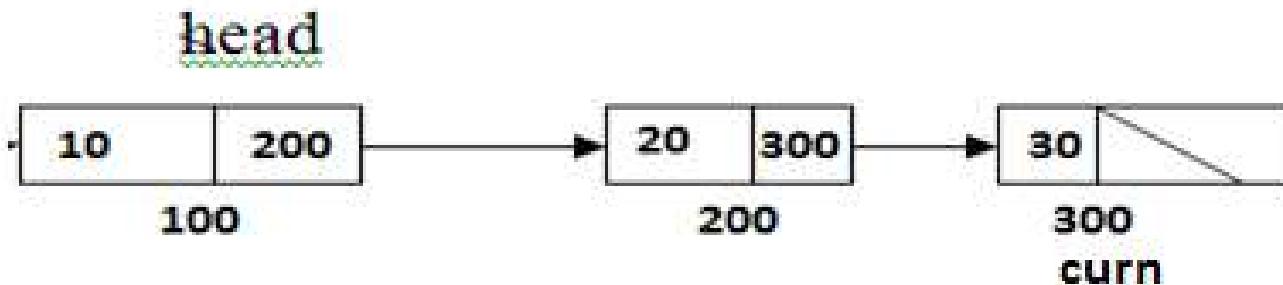
curn-> next is 300 which is not equal to NULL so curn moves to next node

# Insertion at the Rear end



While (curn->next!=NULL)

curn=curn->next;

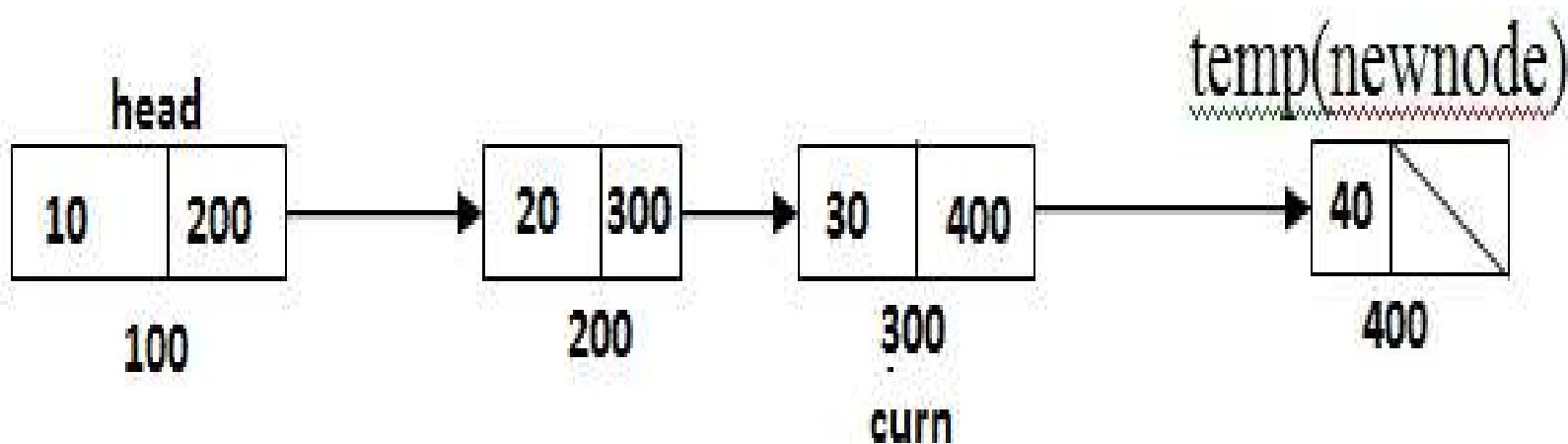


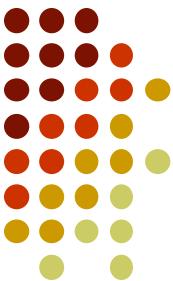
curn-> next is NULL which is not equal to NULL so we stop here

# Insertion at the Rear end



- `curn->next=temp;`

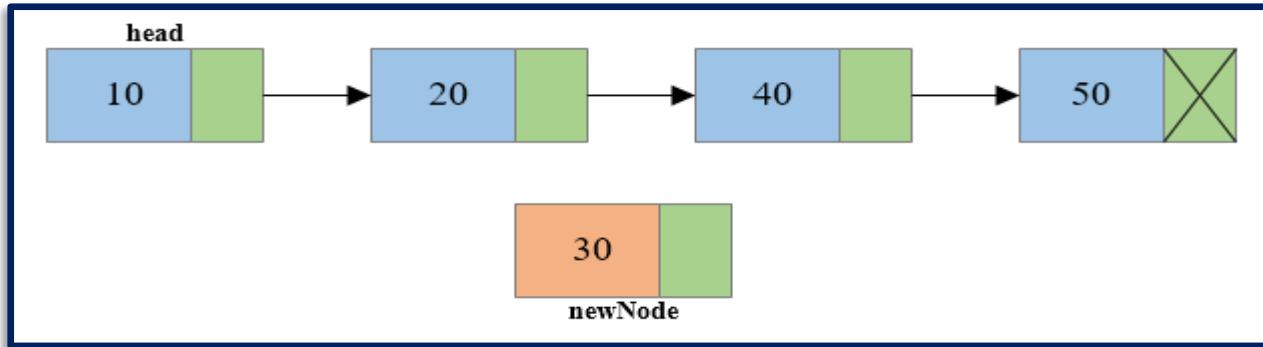




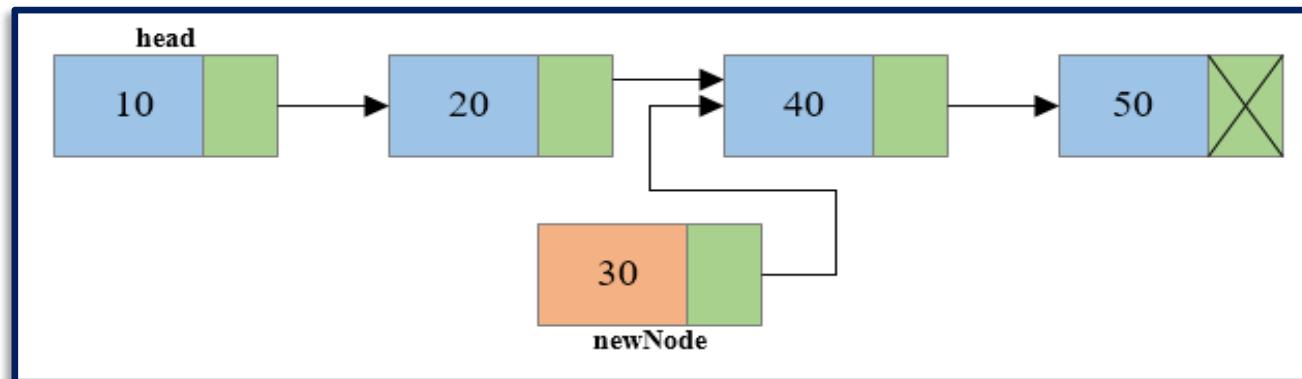
# Insertion at any position

Steps to insert node at any position of Singly Linked List

Step 1: Create a new node.



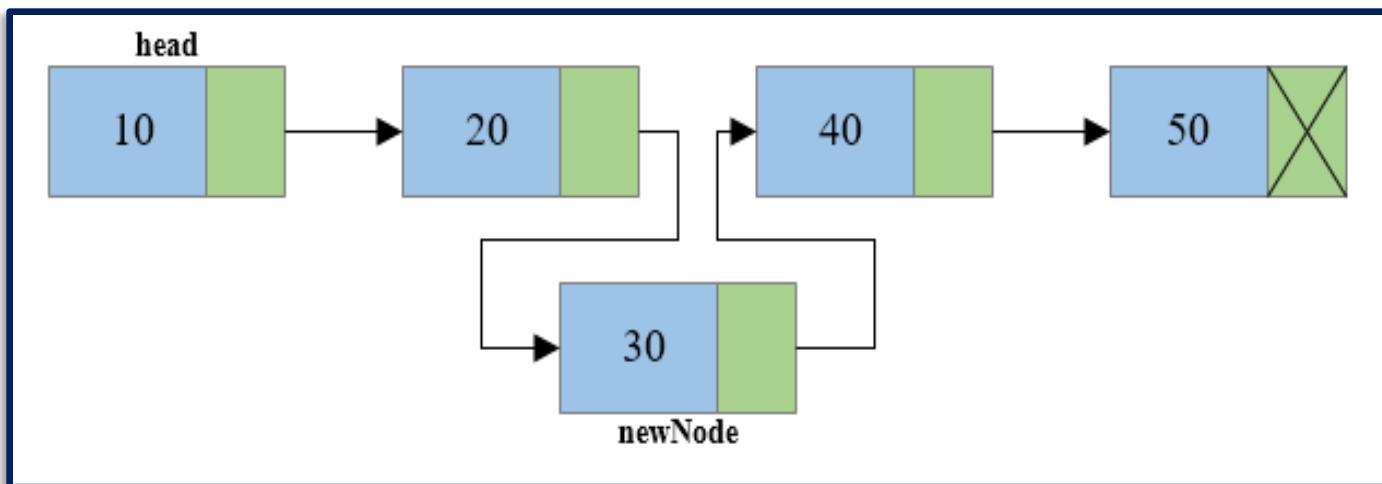
Step 2: Traverse to the  $n-1^{\text{th}}$  position of the linked list and connect the new node with the  $n+1^{\text{th}}$  node. ( $\text{newNode} \rightarrow \text{next} = \text{temp} \rightarrow \text{next}$ ) where temp is the  $n-1^{\text{th}}$  node.



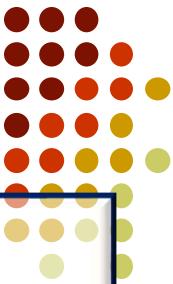


# Insertion at any position

Step 3: Now at last connect the  $n-1^{\text{th}}$  node with the new node i.e. the  $n-1^{\text{th}}$  node will now point to new node. (`temp->next = newNode`) where temp is the  $n-1^{\text{th}}$  node.



# Insertion at any position



```
/* Create a new node and insert at middle of the linked list.*/

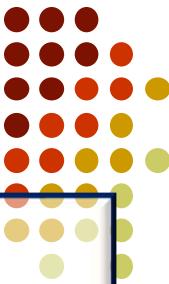
void insertNodeAtMiddle(int data, int position)
{
    int i;
    struct node *newNode, *temp;

    newNode = (struct node*)malloc(sizeof(struct node));

    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        newNode->data = data;      //Links the data part
        newNode->next = NULL;

        temp = head;
```

# Insertion at any position

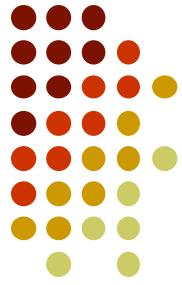


```
for(i=2; i<=position-1; i++) /* Traverse to the n-1 position */
{
    temp = temp->next;

    if(temp == NULL)
        break;
}
if(temp != NULL)
{
    /* Links the address part of new node */
    newNode->next = temp->next;

    /* Links the address part of n-1 node */
    temp->next = newNode;

    printf("DATA INSERTED SUCCESSFULLY\n");
}
else
{
    printf("UNABLE TO INSERT DATA AT THE GIVEN POSITION\n");
}
}
```



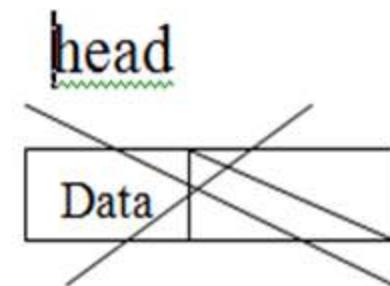
# DELETION AT FRONT END



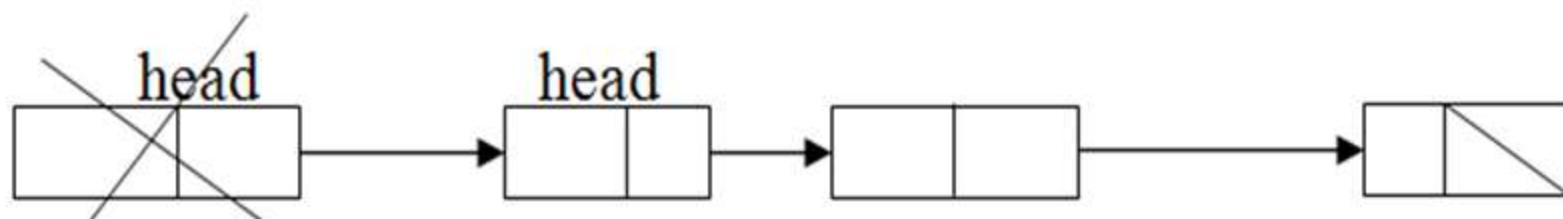
# Deletion at Front end

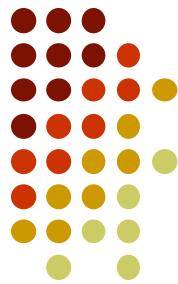
CASE 1: When the list is empty i.e. head=NULL

CASE2: When there is only one node in the list



CASE 3: When there are more than one nodes in the list



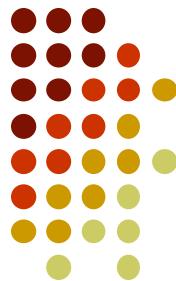


# Deletion at Front end

## CASE-1

1. If the list is empty i.e. head=NULL just display a message that list is empty

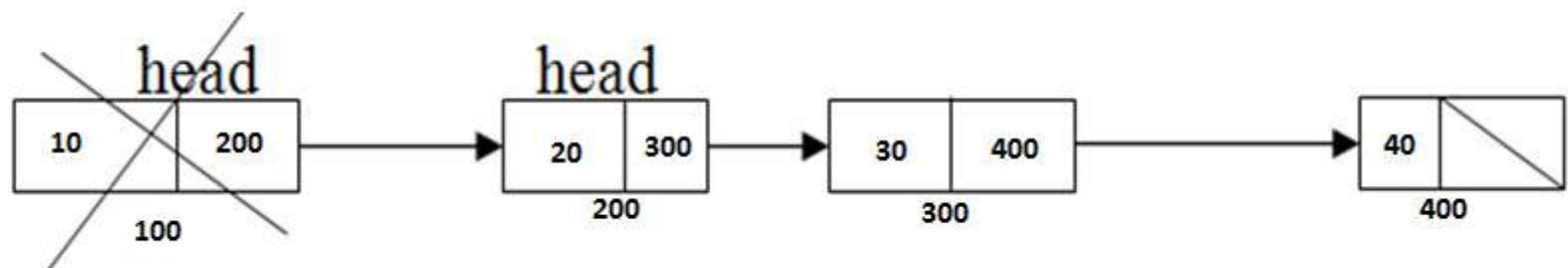
Print list empty



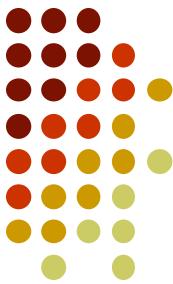
# Deletion at Front end

## CASE-2 & 3

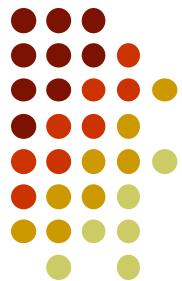
1. Print deleted node is head->info;
2. head=head->next



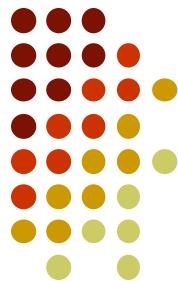
# Deletion at Front end



```
void deletion_front(slist *head)
{
    if(head==NULL) //CASE-1
        printf("LIST IS EMPTY");
    else //CASE 2 & 3
    {
        printf("\nDeleted element is %d",head->info) ;
        head=head->next ;
    }
}
```

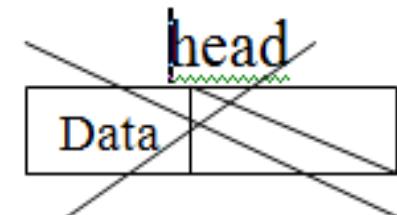


# DELETION AT REAR END



# Deletion at Rear end

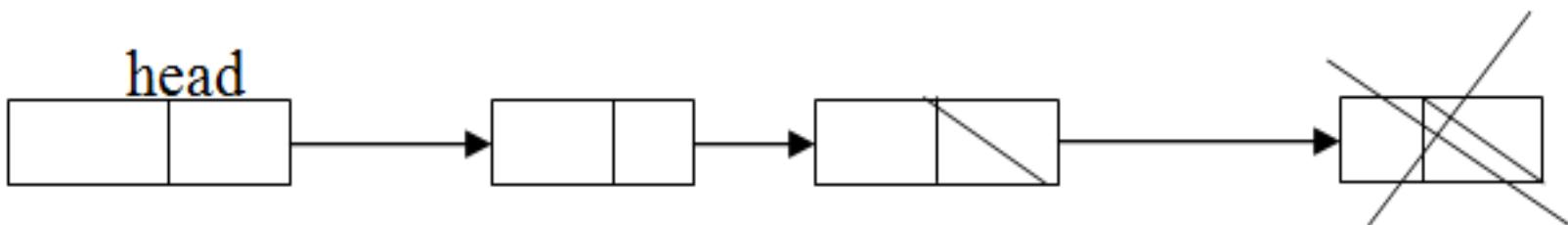
CASE 1: When the list is empty i.e. head=NULL

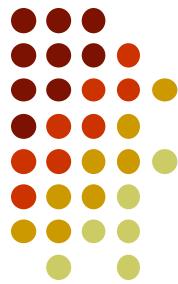


CASE2: When there is only one node in the list

head=NULL

CASE 3: When there are more than one nodes in the list

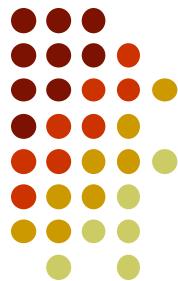




# Deletion at rear end

## CASE-1

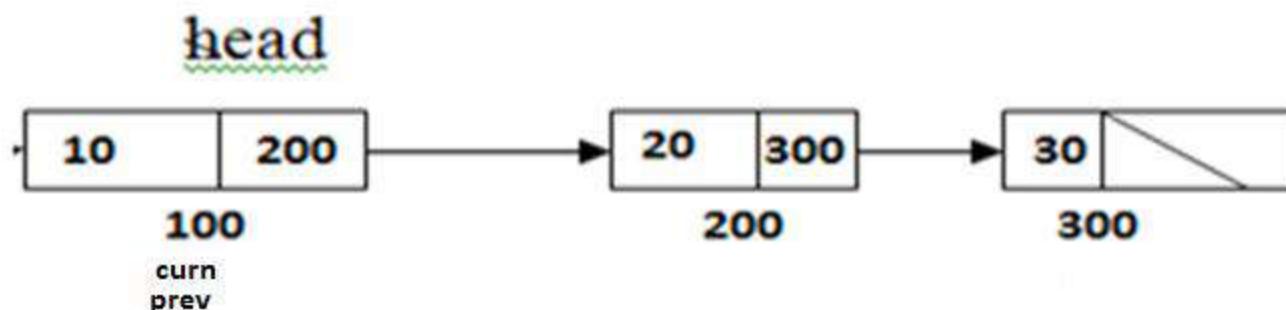
1. if(head==NULL)  
    printf("List is empty");

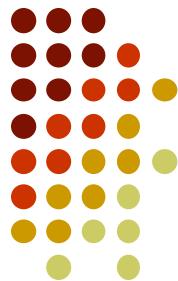


# Deletion at Rear end

## CASE-2 & 3

1. curn=head;
2. prev=curn;

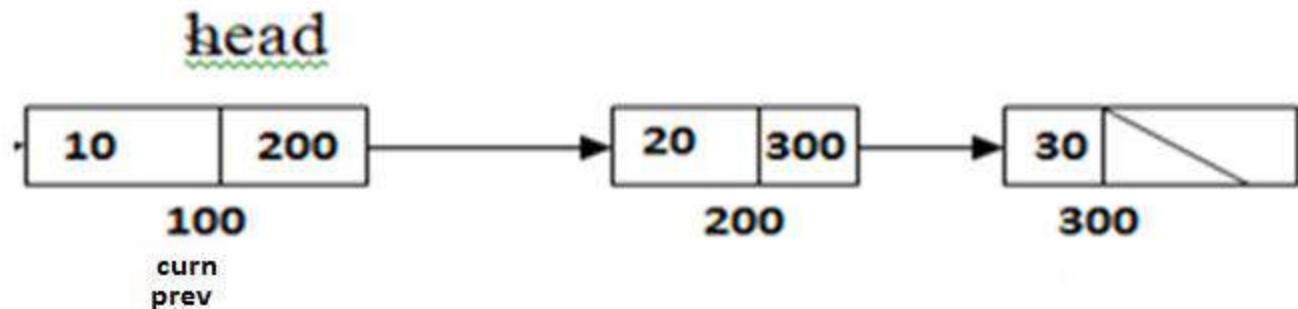




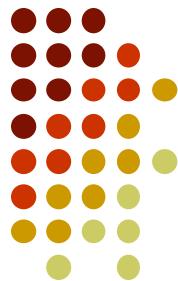
# Deletion at Rear end

## CASE-2 & 3

1. curn=head;
2. prev=curn;
3. while(curn->next !=NULL)
4. {
5. prev=curn;
6. curn=curn->next ;
7. }



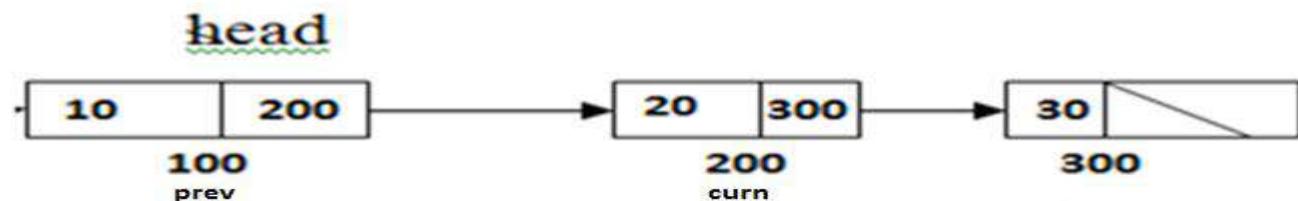
Curn-> next is 200 which is not NULL so we enter into while loop and curn moves to next node



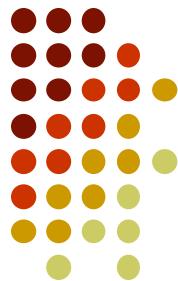
# Deletion at Rear end

## CASE-2 & 3

1. curn=head;
2. prev=curn;
3. while(curn->next !=NULL)
4. {
5.     prev=curn;
6.     curn=curn->next ;
7. }



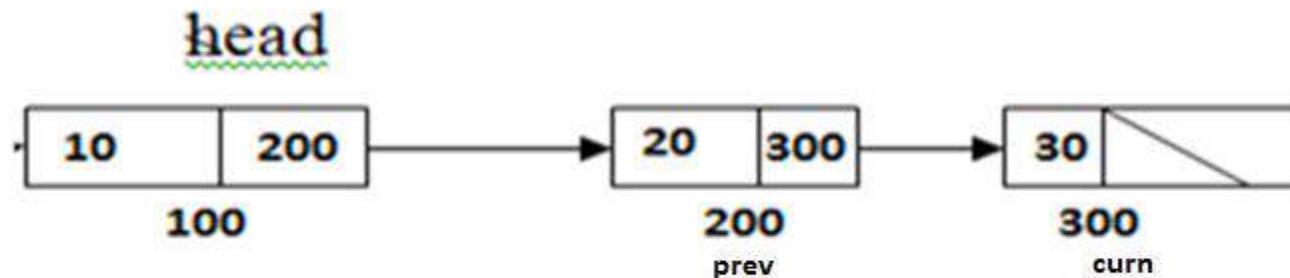
Curn-> next is 300 which is not NULL so curn moves to next node



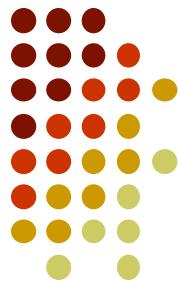
# Deletion at Rear end

## CASE-2 & 3

1. curn=head;
2. prev=curn;
3. while(curn->next !=NULL)
4. {
5.     prev=curn;
6.     curn=curn->next ;
7. }



Now Curn-> next is NULL so we stop

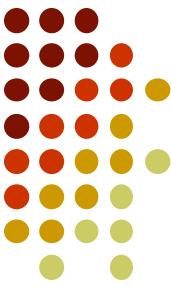


# Deletion at Rear end

## CASE-2 & 3

1. print curn->info deleted
2. delete curn;
3. prev->next =NULL;





# Deletion at Rear end

```
void deletion_rear(slist * head)
{
    slist *curn, *prev;
    if(head==NULL)          // CASE-1
        printf("LIST IS EMPTY");
    else                      // CASE-2 & 3
    {
        curn=head;
        prev=curn;
        while(curn->next!=NULL)
        {
            prev=curn;
            curn=curn->next ;
        }
        printf("\nDeleted element is %d ",curn->info) ;
        delete curn;
        prev->next =NULL;
    }
}
```



# DISPLAY



# Display

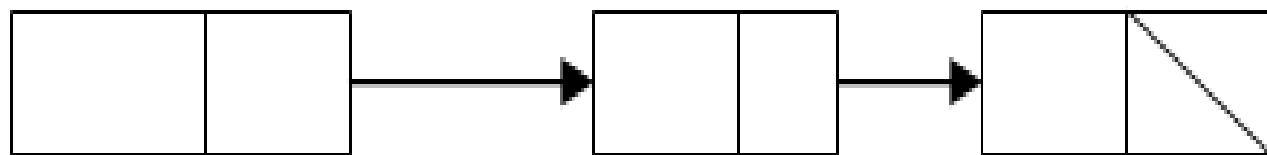
~

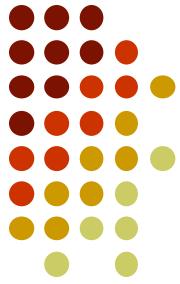
Case 1: When the list is empty

i.e head=NULL

Case 2: When the list is not empty

head





# Display

## CASE-1

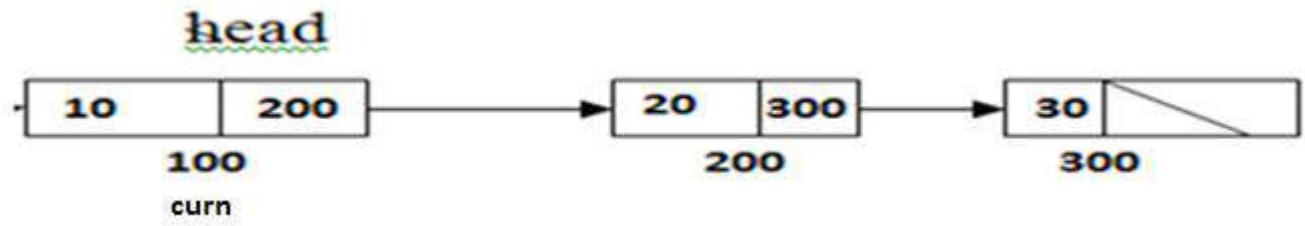
```
1. if(head==NULL)  
    printf("List is empty");
```



# Display

## CASE-2

```
1. else  
2. {     curn=head;  
3.     while(curn!=NULL)  
4.     {  
5.         Printf("\t%d",curn->info );  
6.         curn =curn->next ;  
7.     }  
8. }  
9. }
```



Curn is 100 its not equal to NULL, so curn->info that's 10 is displayed and curn moves to next node

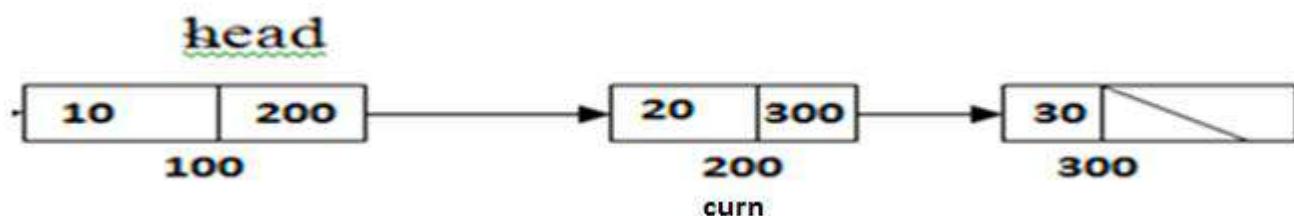
**Output: 10**



# Display

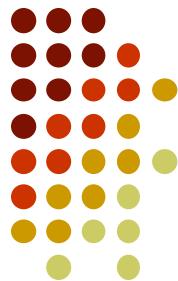
## CASE-2

```
1. else  
2. {     curn=head;  
3.     while(curn !=NULL)  
4.     {  
5.         Printf("\t%d",curn->info );  
6.         curn =curn->next ;  
7.     }  
8. }  
9. }
```



Curn is 200 its not equal to NULL, so curn->info that's 20 is displayed and curn moves to next node

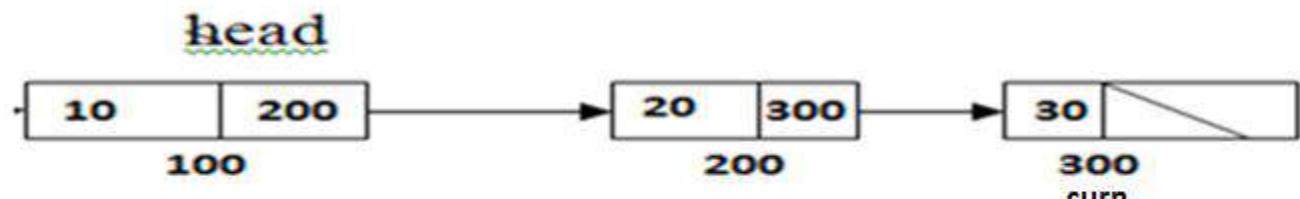
**Output:** 10    20



# Display

## CASE-2

```
1. else  
2. {     curn=head;  
3.     while(curn !=NULL)  
4.     {  
5.         Printf("\t%d",curn->info );  
6.         curn =curn->next ;  
7.     }  
8. }  
9. }
```



Curn is 300 its not equal to NULL, so curn->info that's 30 is displayed and curn moves to next node

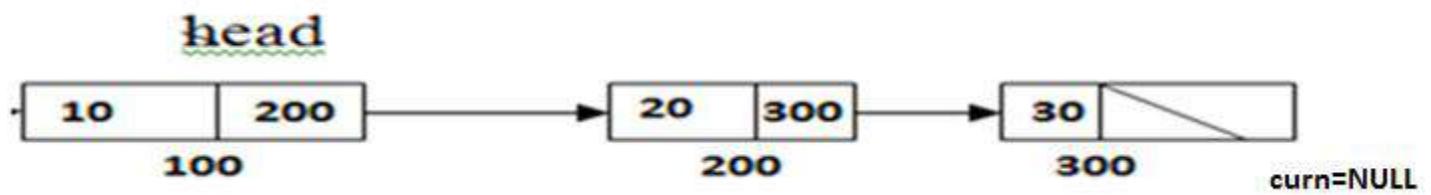
**Output:** 10    20    30



# Display

## CASE-2

```
1. else  
2. {     curn=head;  
3.     while(curn !=NULL)  
4.     {  
5.         Printf("\t%d",curn->info );  
6.         curn =curn->next ;  
7.     }  
8. }  
9. }
```



Now Curn is NULL so while is terminated.

Output: 10 20 30

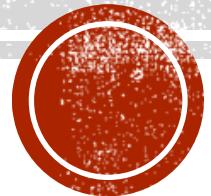


# Display

```
void display(slist *head)
{
    slist * curn;
    if(head==NULL)    // CASE-1
        printf("LIST IS EMPTY");
    else              // CASE-2
    {
        curn=head;
        while(curn !=NULL)
        {
            printf("\t%d",curn->info );
            curn =curn->next ;
        }
    }
}
```

# DATA STRUCTURES AND ALGORITHMS (BCSE202L)

*Module 2: List*

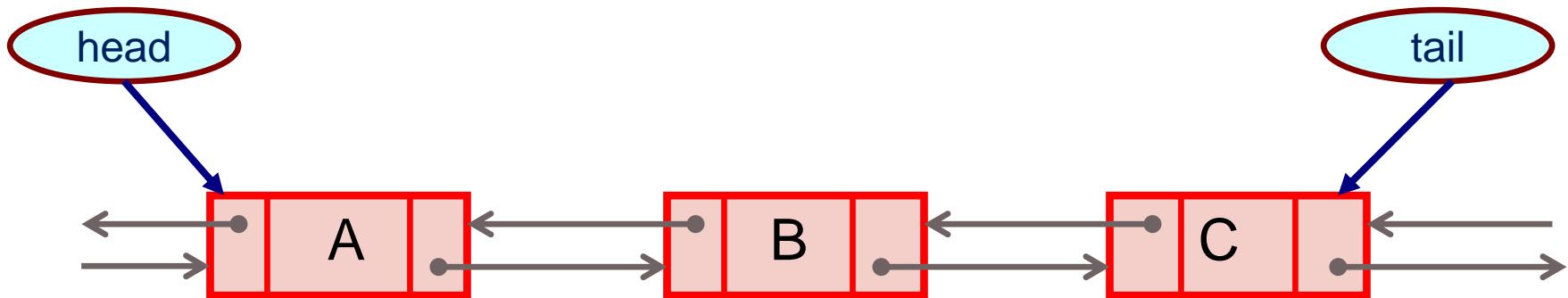


**Dr. Priyanka N**

# TYPES OF LISTS: DOUBLY LINKED LIST

## Doubly linked list

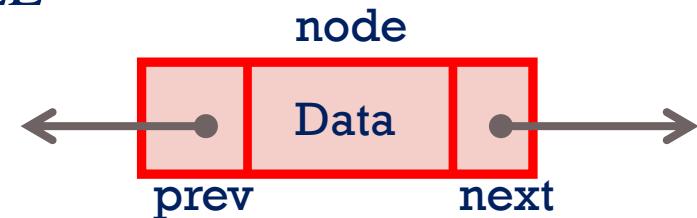
- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, *head* and *tail*.



# DEFINING A NODE OF A DOUBLY LINKED LIST

Each node of doubly linked list (DLL) consists of three fields:

- Item (or) Data
- Pointer of the next node in DLL
- Pointer of the previous node in DLL

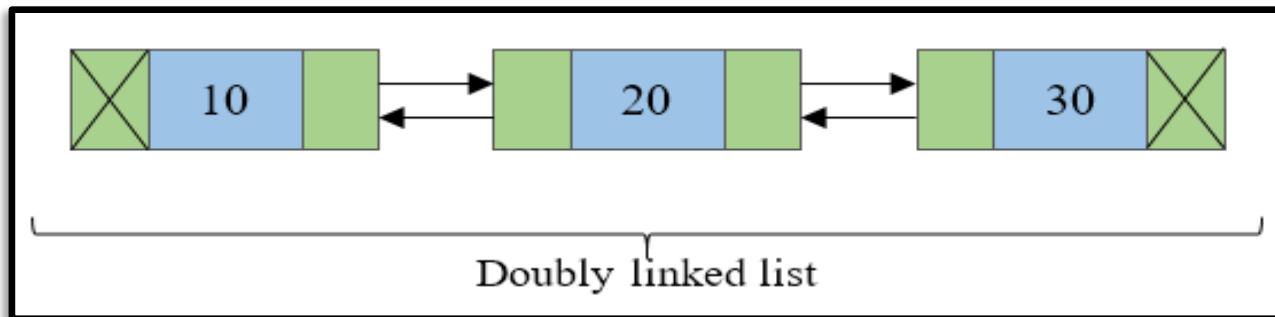


## How to define a node of a doubly linked list (DLL)?

```
struct node
{
    int data;
    struct node *next; // Pointer to next node in DLL
    struct node *prev; // Pointer to previous node in DLL
};
```

# DOUBLY LINKED LIST

- Doubly linked list is a collection of nodes linked together in a sequential way.
- Doubly linked list is almost similar to singly linked list except it contains **two address or reference fields**, where one of the address field contains reference of the next node and other contains reference of the previous node.
- **First and last node** of a linked list contains a terminator generally a **NULL** value, that determines the start and end of the list.
- Doubly linked list is sometimes also referred as **bi-directional linked list** since it allows traversal of nodes in both direction.
- Since doubly linked list allows the traversal of nodes in both direction, we can keep track of both first and last nodes.



# DOUBLE VERSUS SINGLE LINKED LIST

## **Advantages over singly linked list**

- 1) A DLL can be traversed in both forward and backward direction.
- 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

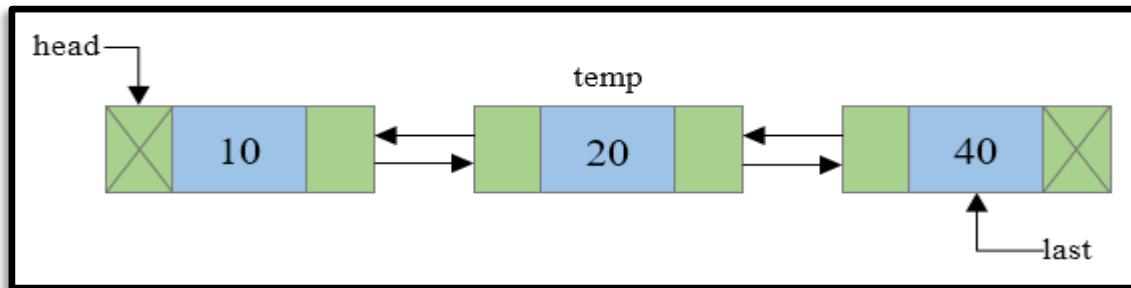
## **Disadvantages over singly linked list**

- 1) Every node of DLL require extra space for an previous pointer.
- 2) All operations require an extra pointer previous to be maintained.

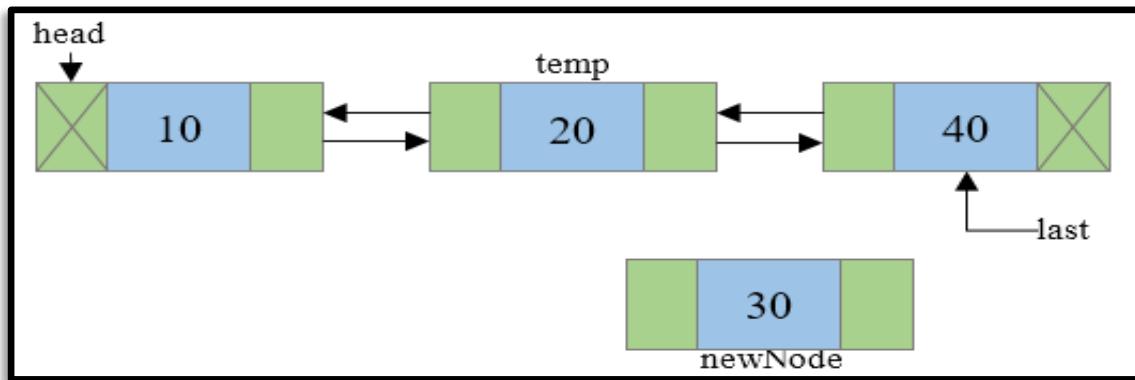
# DOUBLY LINKED LIST: INSERTION AT ANY POSITION

Steps to insert a new node at  $n^{\text{th}}$  position in a Doubly linked list.

Step 1: Traverse to  $N-1$  node in the list, where  $N$  is the position to insert. Say **temp** now points to  $N-1^{\text{th}}$  node.

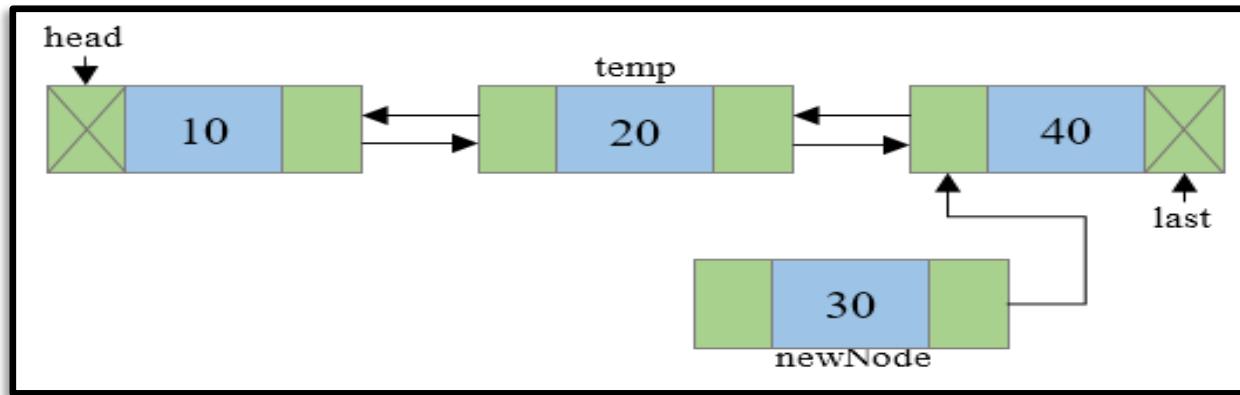


Step 2: Create a **newNode** that is to be inserted and assign some data to its data field.

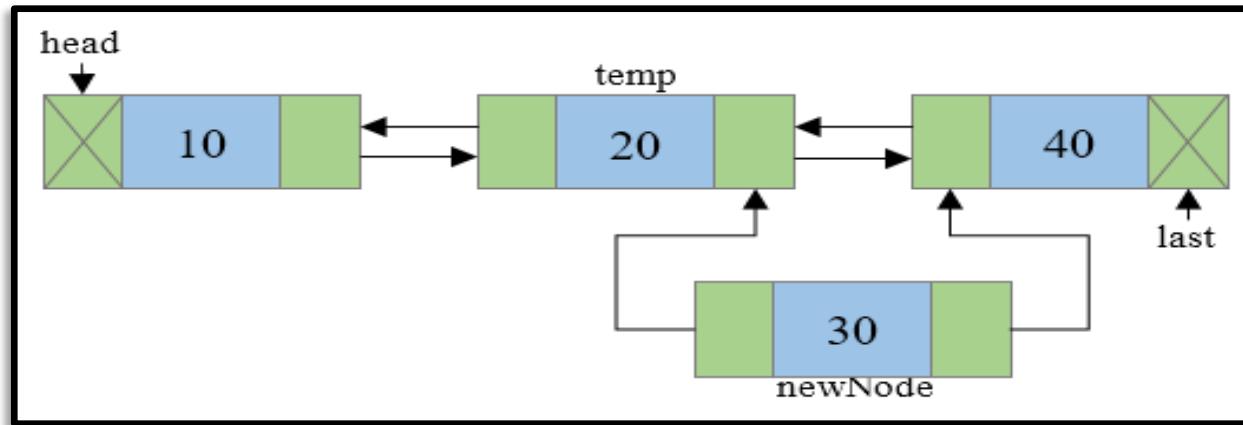


# DOUBLY LINKED LIST: INSERTION AT ANY POSITION

**Step 3:** Connect the next address field of **newNode** with the node pointed by next address field of **temp** node.

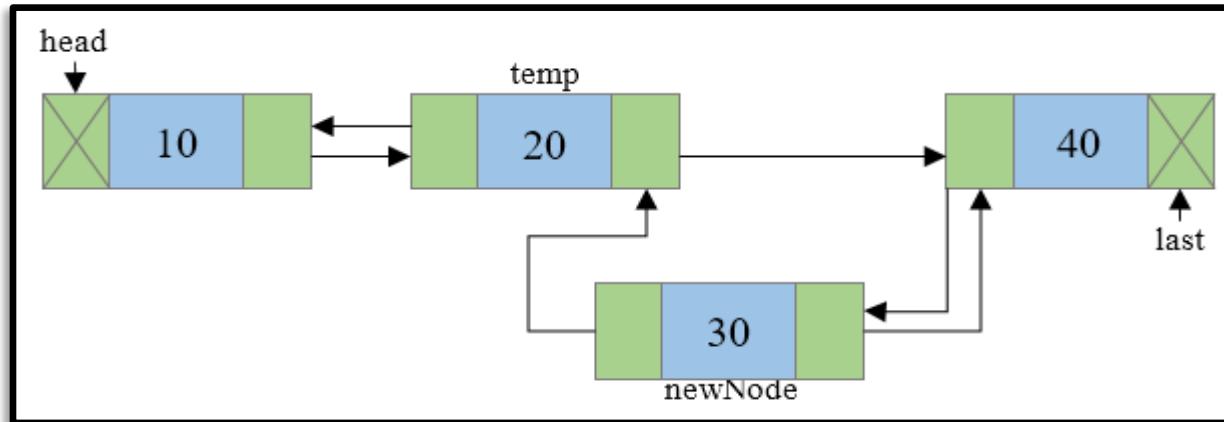


**Step 4:** Connect the previous address field of **newNode** with the **temp** node.

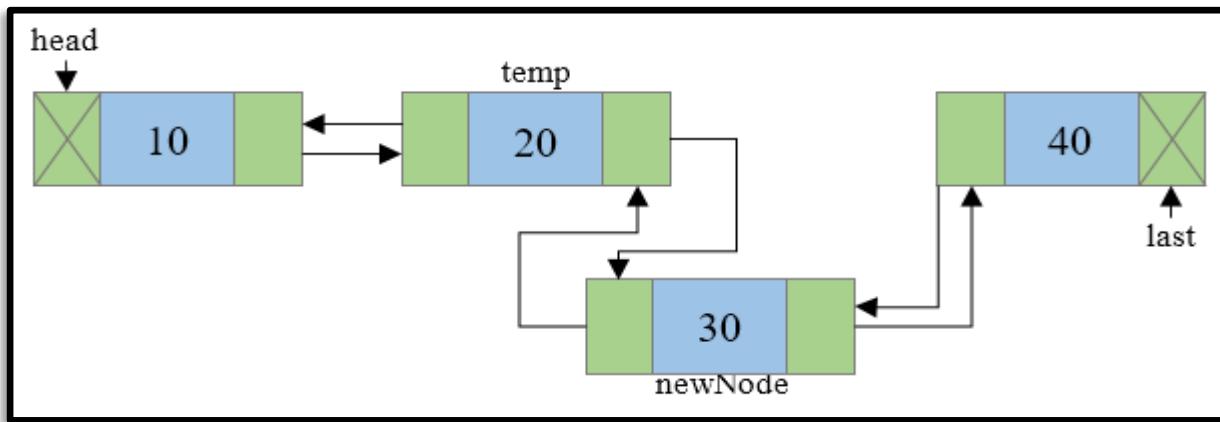


# DOUBLY LINKED LIST: INSERTION AT ANY POSITION

**Step 5:** Check if `temp->next` is not `NULL` then, connect the previous address field of node pointed by `temp->next` to `newNode`.

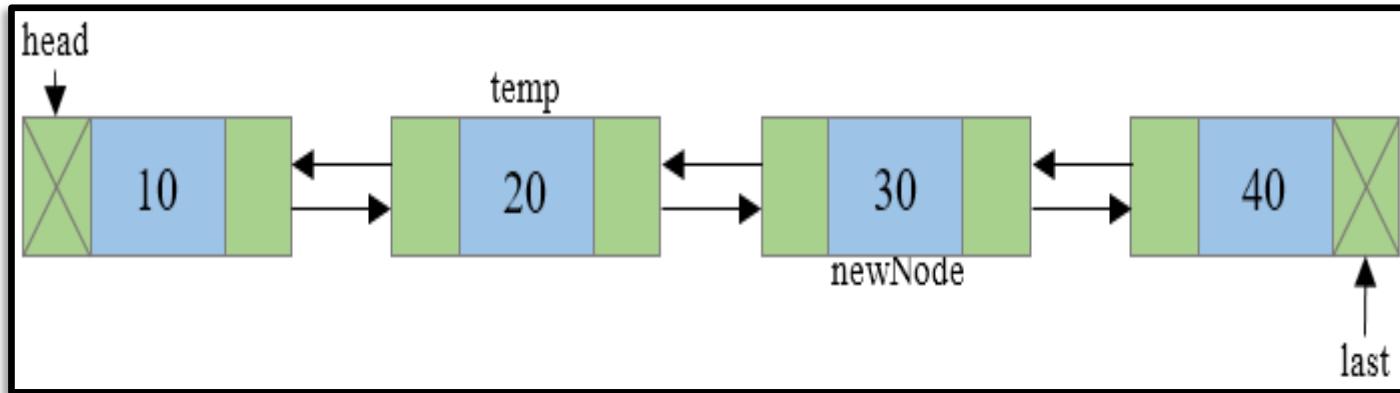


**Step 6:** Connect the next address field of `temp` node to `newNode`.



# DOUBLY LINKED LIST: INSERTION AT ANY POSITION

Step 7: Final doubly linked list looks like



# DOUBLY LINKED LIST: INSERTION AT ANY POSITION

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

struct node {                  /* Basic structure of Node */
    int data;
    struct node * prev;
    struct node * next;
}*head, *last;

int main()
{
    int n, data;
    head = NULL;
    last = NULL;

    printf("Enter the total number of nodes in list: ");
    scanf("%d", &n);
    createList(n);                // function to create double linked list
    displayList();                // function to display the list

    printf("Enter the position and data to insert new node: ");
    scanf("%d %d", &n, &data);
    insert_position(data, n);     // function to insert node at any position
    displayList();
    return 0;
}
```

# DOUBLY LINKED LIST: INSERTION AT ANY POSITION

```
void createList(int n)
{
    int i, data;
    struct node *newNode;
    if(n >= 1) /* Creates and links the head node */
        head = (struct node *)malloc(sizeof(struct node));
        printf("Enter data of 1 node: ");
        scanf("%d", &data);
        head->data = data;
        head->prev = NULL;
        head->next = NULL;

        last = head;

        for(i=2; i<=n; i++) /* Creates and links rest of the n-1 nodes */
            newNode = (struct node *)malloc(sizeof(struct node));
            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->prev = last; //Links new node with the previous node
            newNode->next = NULL;

            last->next = newNode; //Links previous node with the new node
            last = newNode; //Makes new node as last/previous node
    }
    printf("\nDOUBLY LINKED LIST CREATED SUCCESSFULLY\n");
}
```

# DOUBLY LINKED LIST: INSERTION AT ANY POSITION

```
void insert_position(int data, int position)
{
    struct node * newNode, *temp;
    if(head == NULL) {
        printf("Error, List is empty!\n");
    }
    else{
        temp = head;
        if(temp!=NULL) {
            newNode = (struct node *)malloc(sizeof(struct node));

            newNode->data = data;
            newNode->next = temp->next; //Connects new node with n+1th node
            newNode->prev = temp;           //Connects new node with n-1th node

            if(temp->next != NULL)
            {
                temp->next->prev = newNode; /* Connects n+1th node with new node */
            }
            temp->next = newNode;           /* Connects n-1th node with new node */
            printf("NODE INSERTED SUCCESSFULLY AT %d POSITION\n", position);
        }
        else{
            printf("Error, Invalid position\n");
        }
    }
}
```

# DOUBLY LINKED LIST: INSERTION AT ANY POSITION

```
void displayList()
{
    struct node * temp;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        temp = head;
        printf("DATA IN THE LIST:\n");

        while(temp != NULL)
        {
            printf("DATA of %d node = %d\n", n, temp->data);
            n++;

            /* Moves the current pointer to next node */
            temp = temp->next;
        }
    }
}
```

# FEW EXERCISES TO TRY OUT

**For doubly linked list write a function to:**

- Insert a node at front of the list and at end of the list.

`insert_front (data) ;`

`insert_end (data) ;`

`delete_front (data) ;`

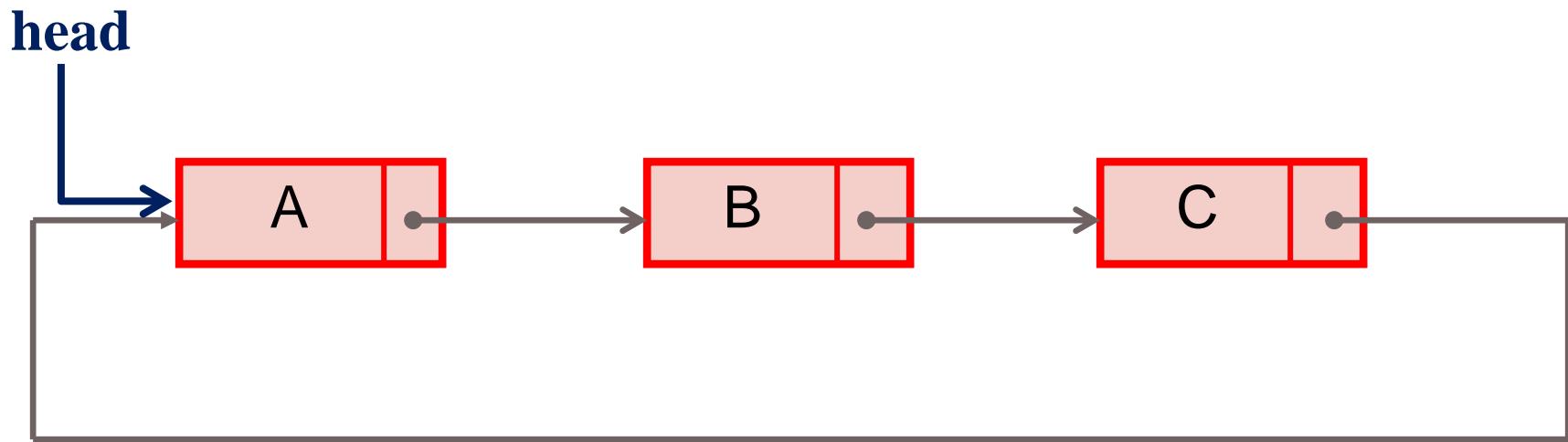
`delete_end (data) ;`

- Sort the DLL in ascending order.
- Count the number of nodes in the given DLL.

# TYPES OF LISTS: CIRCULAR LINKED LIST

## Circular linked list

- The pointer from the last element in the list points back to the first element.

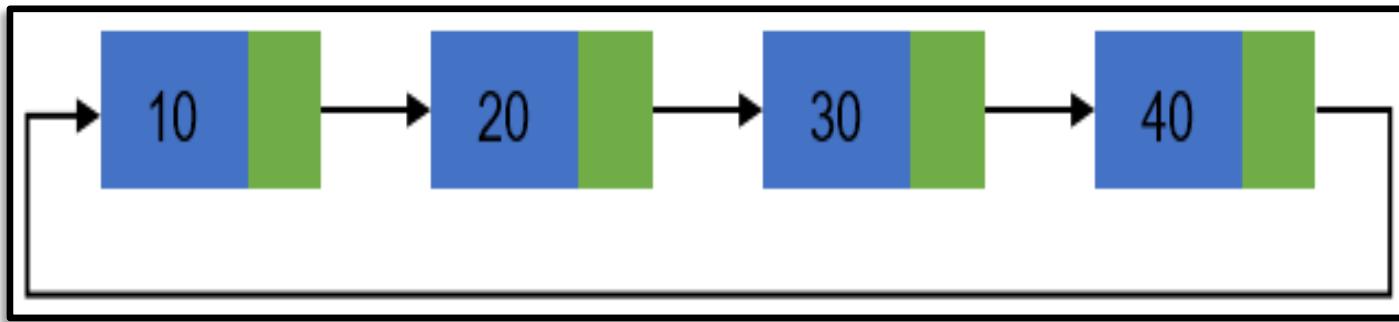


# CIRCULAR LINKED LIST

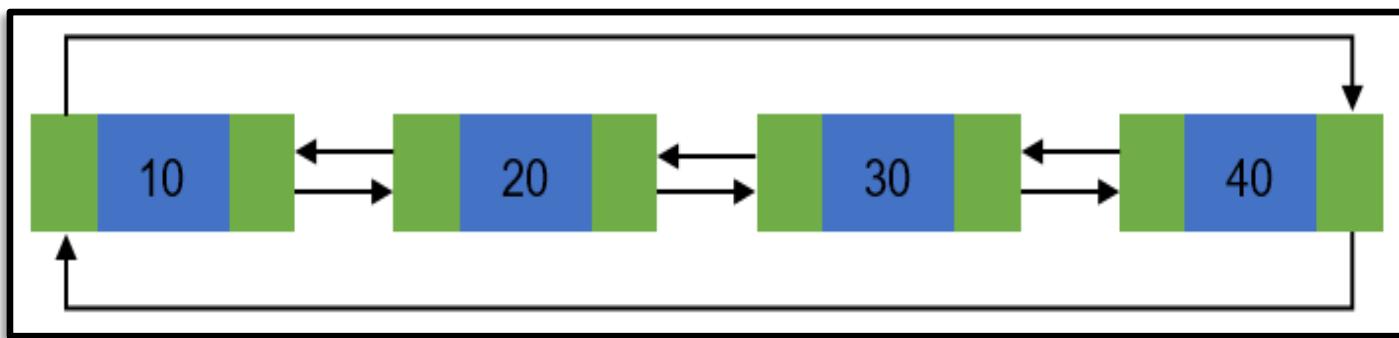
- A circular linked list is basically a linear linked list that may be **single-** or **double-linked**.
- The only difference is that **there is no any NULL** value terminating the list.
- In fact in the list every node points to the next node and last node points to the first node, thus forming a circle. Since it forms a **circle with no end to stop** it is called as **circular linked list**.
- In circular linked list there can be no starting or ending node, whole node can be **traversed from any node**.
- In order to traverse the circular linked list, only once we need to traverse entire list until the **starting node is not traversed again**.
- A circular linked list can be implemented using both **singly linked list** and **doubly linked list**.

# CIRCULAR LINKED LIST:

Basic structure of singly circular linked list:



Doubly circular linked list:



# CIRCULAR LINKED LIST:

## Advantages of a Circular linked list

- Entire list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- Despite of being singly circular linked list we can easily traverse to its previous node, which is not possible in singly linked list.

## Disadvantages of Circular linked list

- Circular list are complex as compared to singly linked lists.
- Reversing of circular list is complex as compared to singly or doubly lists.
- If not traversed carefully, then we could end up in an infinite loop.
- Like singly and doubly lists circular linked lists also doesn't supports direct accessing of elements.

# OPERATIONS ON CIRCULAR LINKED LIST

- Creation of list
- Traversal of list
- Insertion of node
  - At the beginning of list
  - At any position in the list
- Deletion of node
  - Deletion of first node
  - Deletion of node from middle of the list
  - Deletion of last node
- Counting total number of nodes
- Reversing of list

# CREATION AND TRAVERSAL OF A CIRCULAR LIST

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

/* Basic structure of Node */

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

int main()
{
    int n, data;
    head = NULL;

    printf("Enter the total number of nodes in list: ");
    scanf("%d", &n);
    createList(n);                      // function to create circular linked list
    displayList();                      // function to display the list

    return 0;
}
```

# CIRCULAR LINKED LIST: CREATION OF LIST

```
void createList(int n)
{
    int i, data;
    struct node *prevNode, *newNode;
    if(n >= 1){                         /* Creates and links the head node */
        head = (struct node *)malloc(sizeof(struct node));

        printf("Enter data of 1 node: ");
        scanf("%d", &data);

        head->data = data;
        head->next = NULL;
        prevNode = head;

        for(i=2; i<=n; i++){           /* Creates and links rest of the n-1 nodes */
            newNode = (struct node *)malloc(sizeof(struct node));

            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->next = NULL;
            prevNode->next = newNode; //Links the previous node with newly created node
            prevNode = newNode;      //Moves the previous node ahead
        }
        prevNode->next = head; //Links the last node with first node
        printf("\nCIRCULAR LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
```

# CIRCULAR LINKED LIST: TRAVERSAL OF LIST

```
void displayList()
{
    struct node *current;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        current = head;
        printf("DATA IN THE LIST:\n");

        do {
            printf("Data %d = %d\n", n, current->data);

            current = current->next;
            n++;
        }while(current != head);
    }
}
```

# FEW EXERCISES TO TRY OUT

**For circular linked list write a function to:**

- Insert a node at any position of the list and delete from the beginning of the list.

```
insert_position(data,position);  
delete_front();
```

- Reverse the given circular linked link.

# TREES

Dr. Priyanka N

Assistant Professor Senior Grade I

School of Computer Science & Engineering  
VIT, Vellore.

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# TREES

- Introduction to Trees & Terminologies
- Binary Tree – Definition and Properties
- Binary Tree Representation
- Tree Traversals
- Expression Trees: Binary Search Trees
- Operations on Binary Search Trees
  - Insertion
  - Deletion
  - Finding Max and Minimum
  - Finding the  $k^{\text{th}}$  Minimum Element

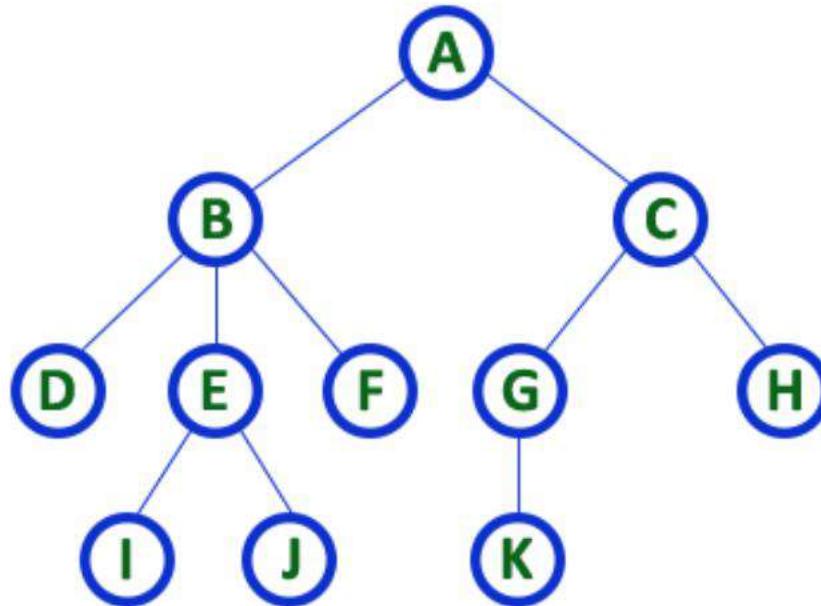
# Trees

- Linear data structure - data is organized in sequential order.
- Non-linear data structure - data is organized in random order
- A tree is a very popular non-linear data structure used in a wide range of applications.
- Tree is a non-linear data structure which organizes data in hierarchical structure
- Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively.

# Tree Data Structure

- Every individual element is called as **Node**
- Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.
- **N** number of nodes then we can have a maximum of **N-1** number of edges.

# Tree - Example



- *Tree with 11 nodes and 10 Edges*
- *In any tree with  $n$  nodes, there will be maximum of  $n-1$  edges*

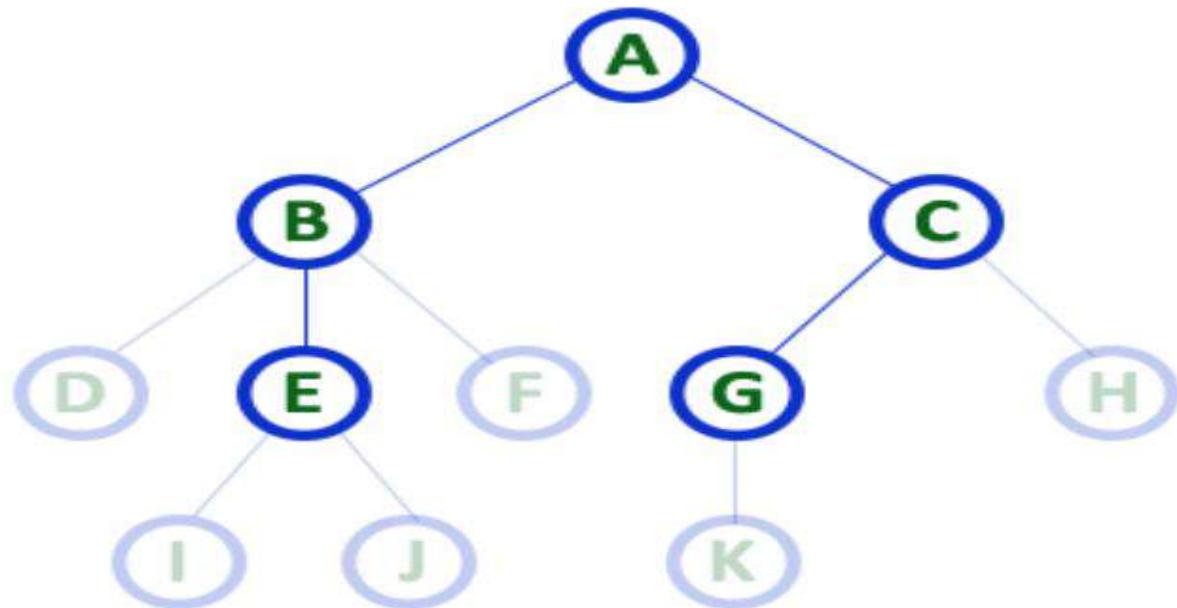
# Formal Definition of Tree

- Defined as the finite set of one or more nodes such that there is a specially designated node called root and the remaining nodes are partitioned into  $n \geq 0$  different sets  $T_1, T_2, T_3, \dots, T_n$  where  $T_1, T_2, T_3, \dots, T_n$  called as Sub Trees.

# Tree Terminologies

- **Root**
  - Root node is the origin of the tree data structure
  - In any tree, there must be only one root node.
- **Edge**
  - The connecting link between any two nodes is called as **EDGE**.
- **Parent**
  - Node which is a predecessor of any node is called as **PARENT NODE** or node which has child / children"

# Parent Nodes

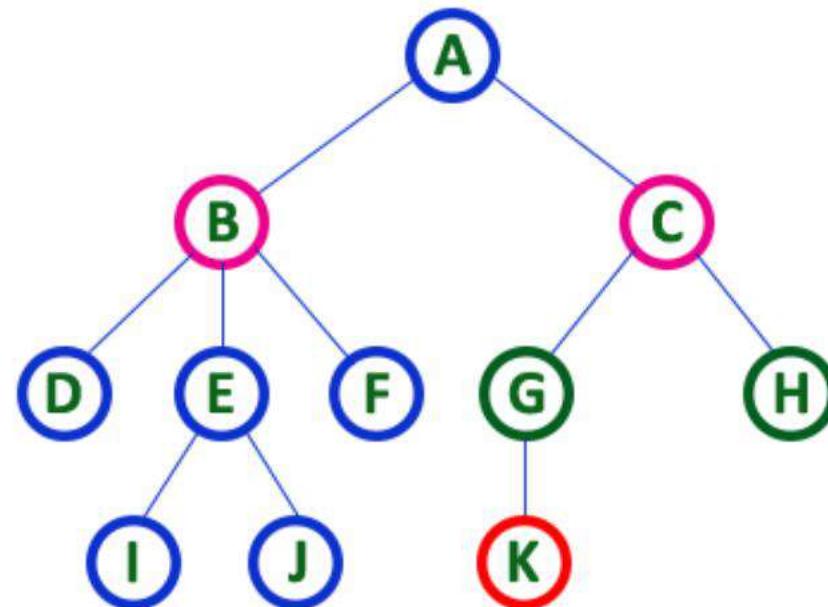


**Child Nodes**-node which is descendant of any node is called as **CHILD Node**.

In simple words, the node which has a link from its parent node is called as child node.

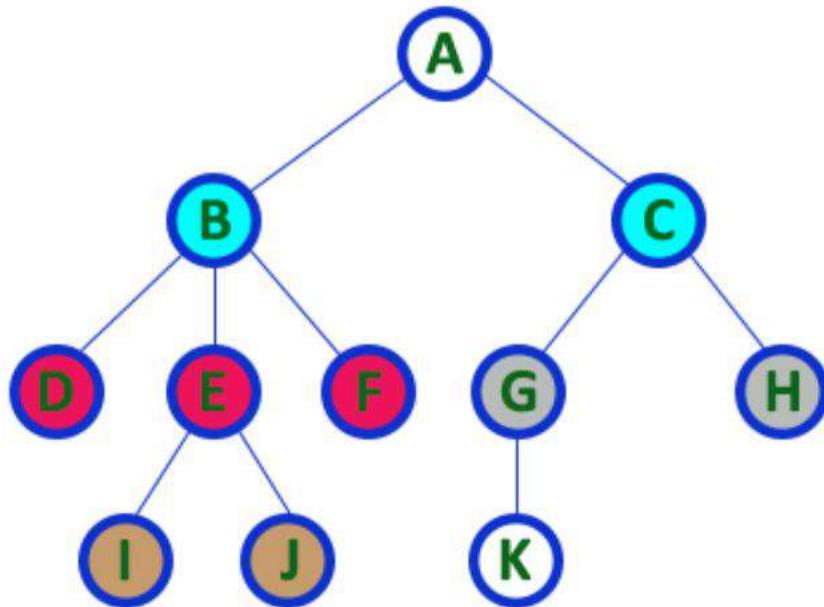
All the nodes except root are child nodes.

B and C are Children of A etc



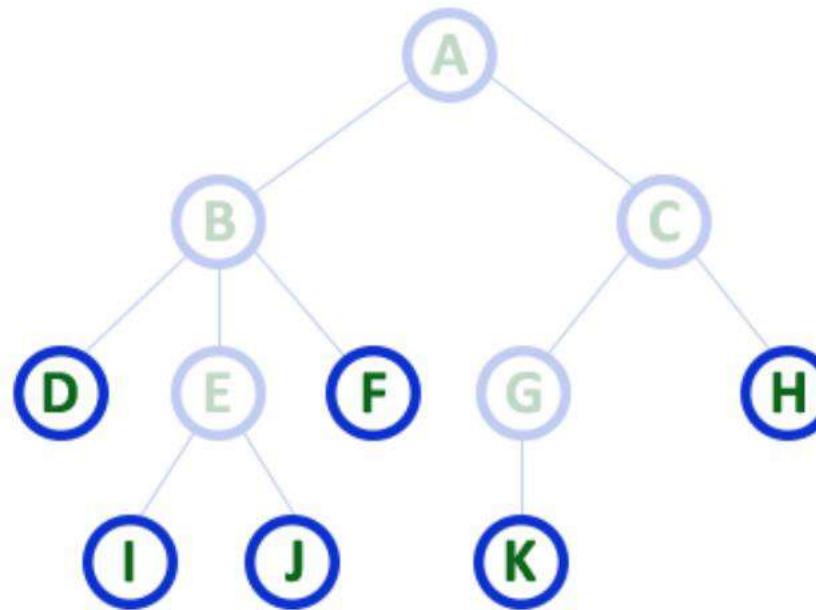
# Siblings

- Nodes which belong to same Parent are called as **SIBLINGS**.
- B and C are Siblings of A etc.....



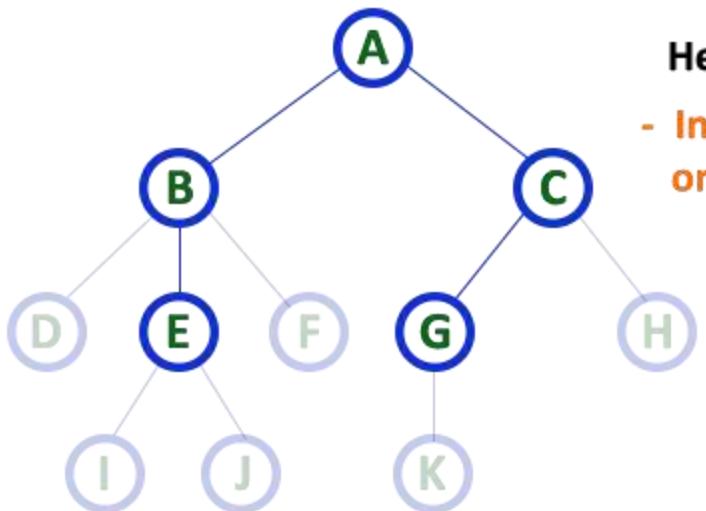
# Leaf/ External Nodes/ Terminal nodes

- Node which does not have a child is called as **LEAF Node**



# Internal Nodes/Non Terminal Nodes

- Node which has at least one child is called as **INTERNAL Node**.

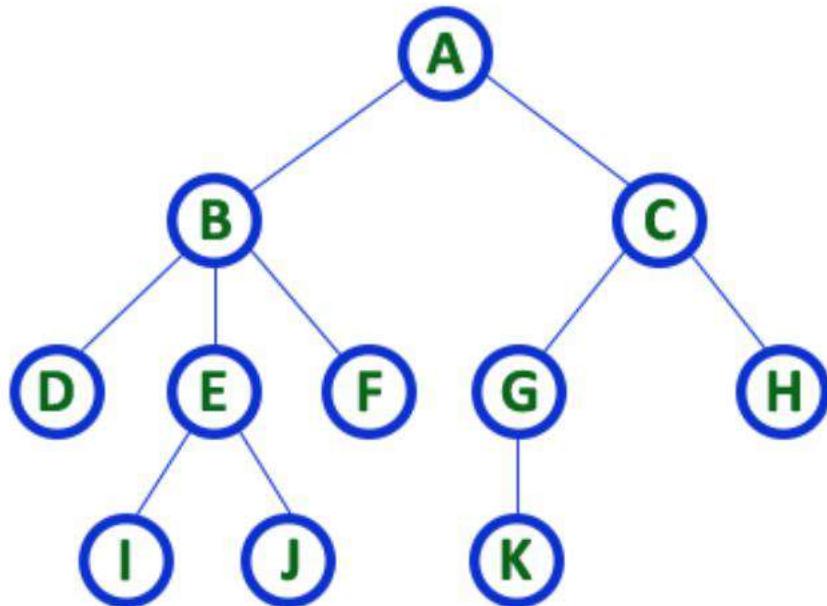


Here A, B, C, E & G are **Internal** nodes

- In any tree the node which has atleast one child is called '**Internal**' node
- Every non-leaf node is called as '**Internal**' node

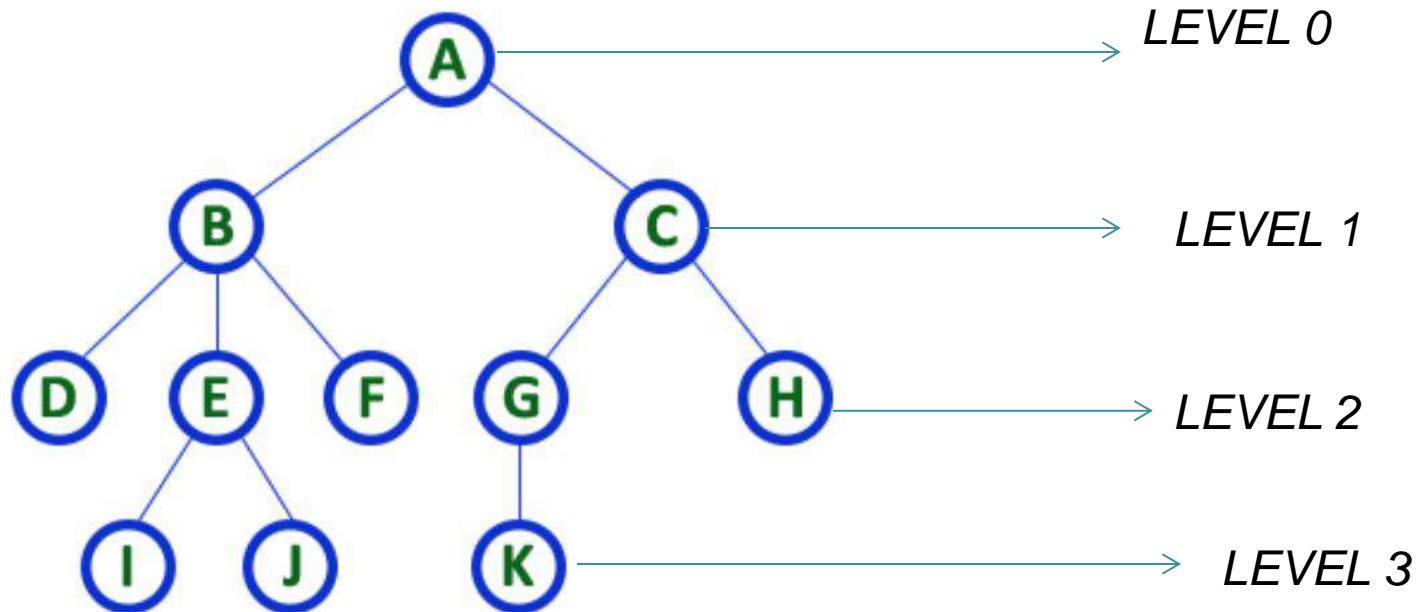
# Degree

- Total number of children of a node is called as **DEGREE** of that Node.



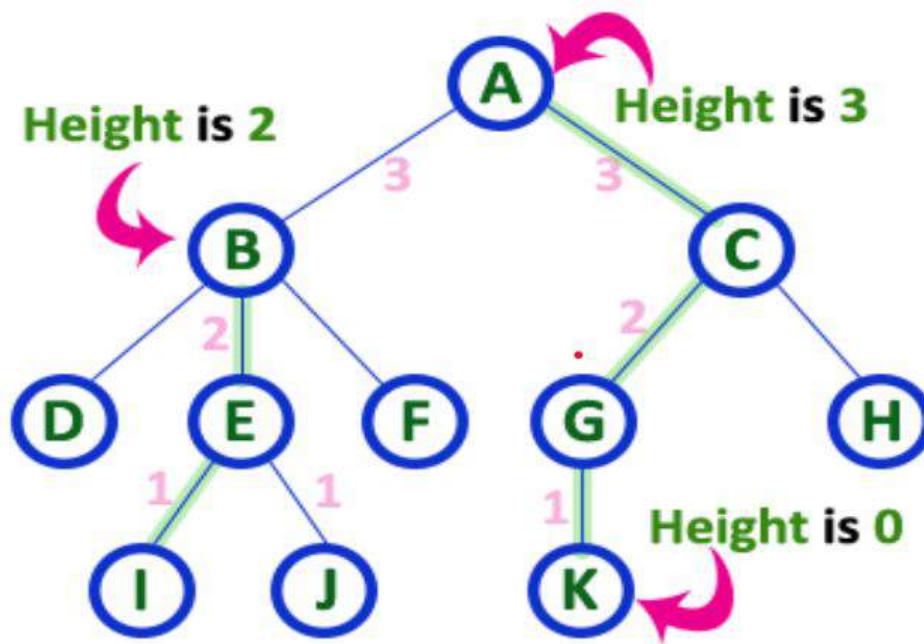
*Here Degree of B is 3  
Here Degree of A is 2*

**Level**-- each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level

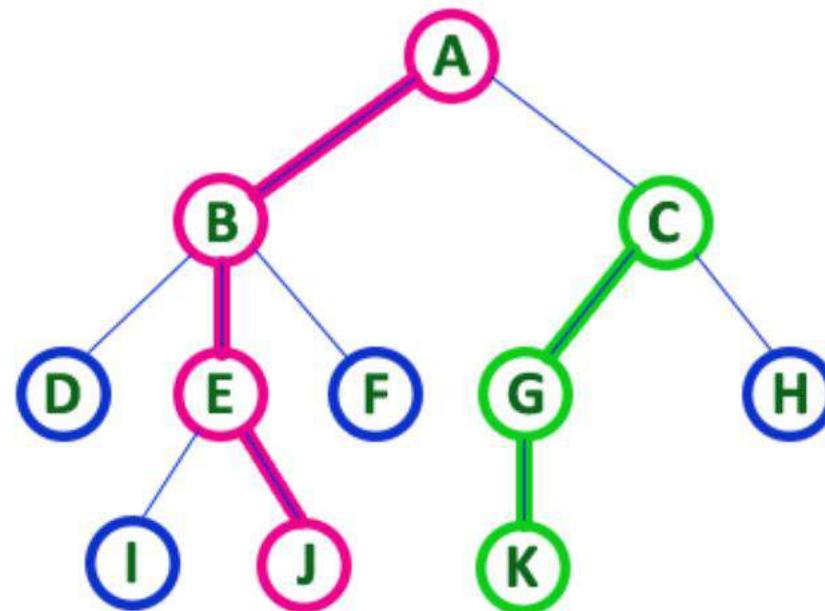


# Height

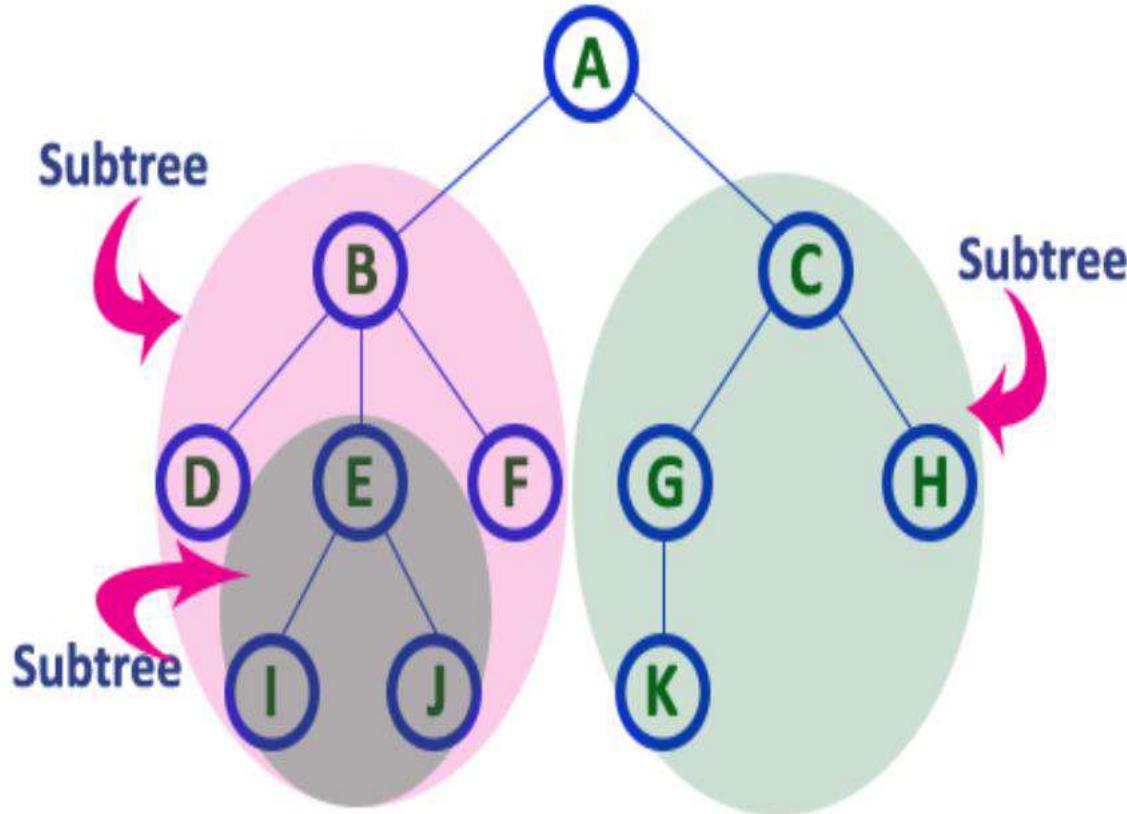
- Total number of edges from leaf node to a particular node in the longest path



Path - sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes



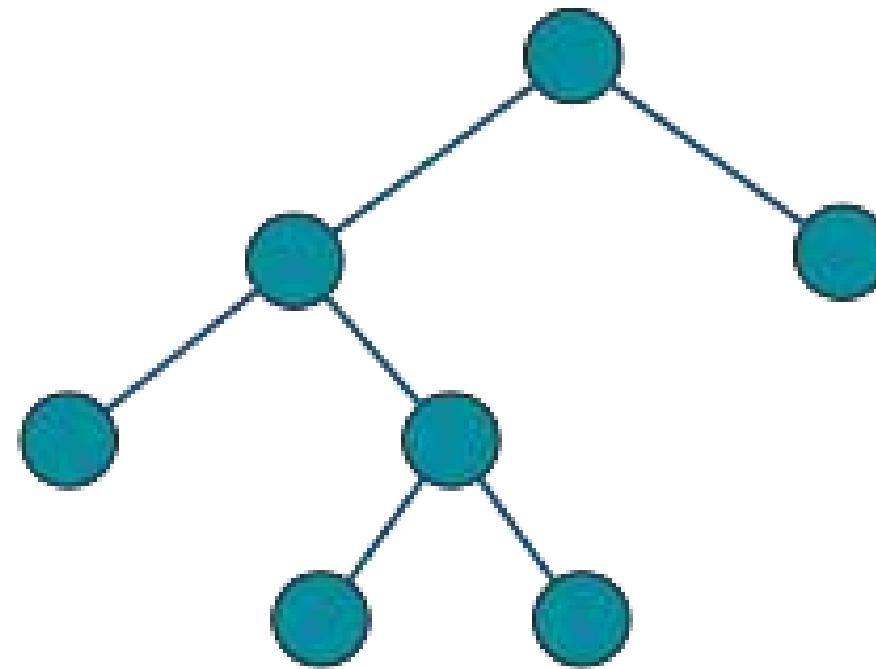
# Sub Tree - Each child from a node forms a subtree recursively



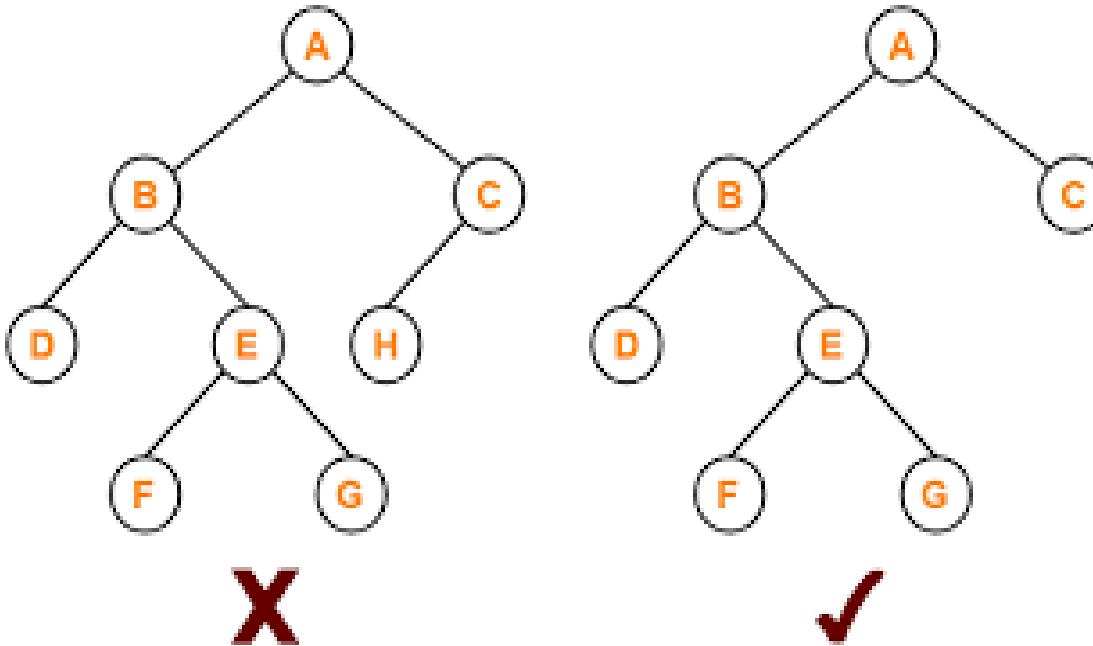
# Binary Tree

- In a normal tree, every node can have any number of children.
- A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**.
- Left child and the other one is Right child
- Every node can have either 0 children or 1 child or 2 children but not more than 2 children

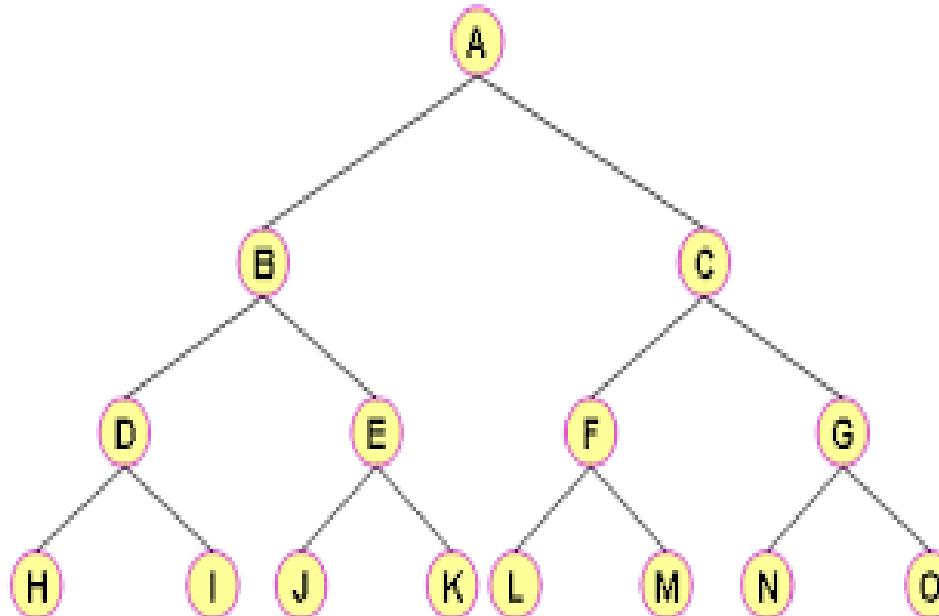
# Binary Tree - Example



# Strictly Binary Tree/Full binary tree/Proper Binary Tree/2-tree



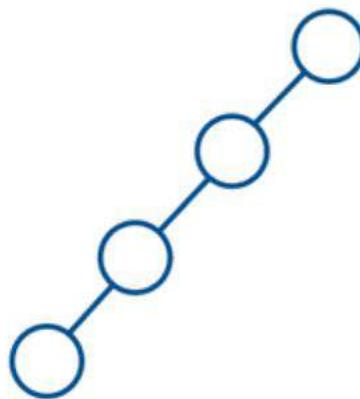
# Complete Binary Tree/Perfect Binary Tree



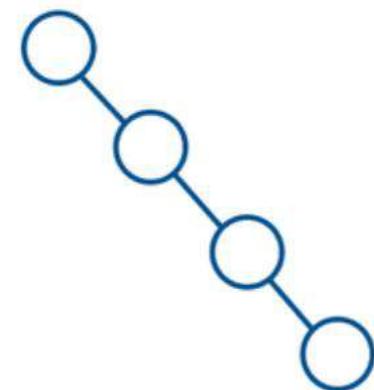
# Skewed Binary Tree

- A skewed binary tree is a type of binary tree in which all the nodes have only either one child or no child.
  - **Left Skewed Binary Tree**
  - **Right Skewed Binary Tree**

# Left Skewed

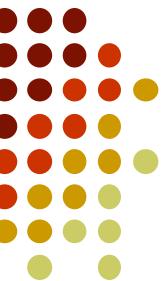


# Right Skewed



# **Binary Tree Representation**

- **Array Representation**
- **Linked List Representation**



# TREES

Dr.Priyanka N



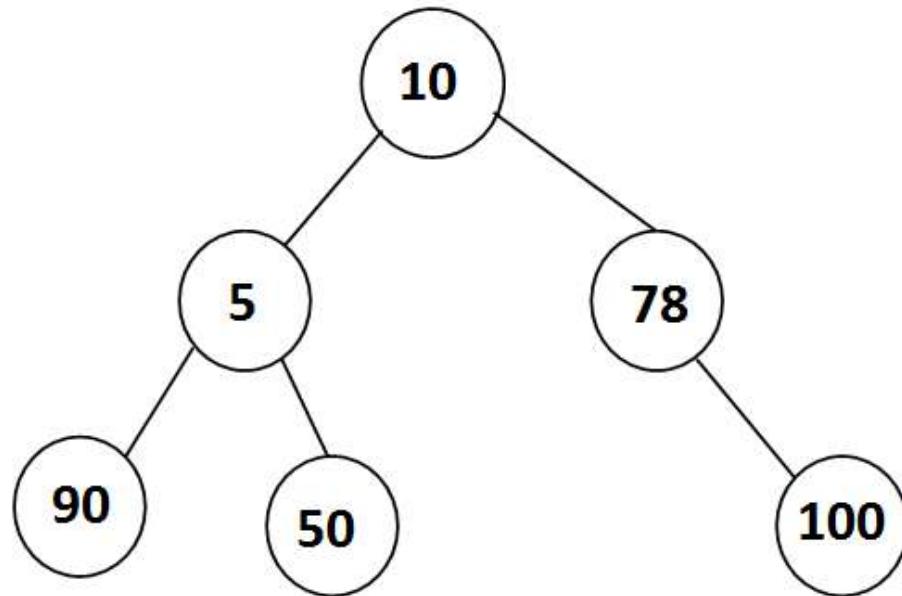
# Traversal

- Traversal is visiting each and every node in the tree.
- There are three types of traversals
  - (i) In Order Traversal
  - (ii) Post Order Traversal
  - (iii) Pre Order Traversal



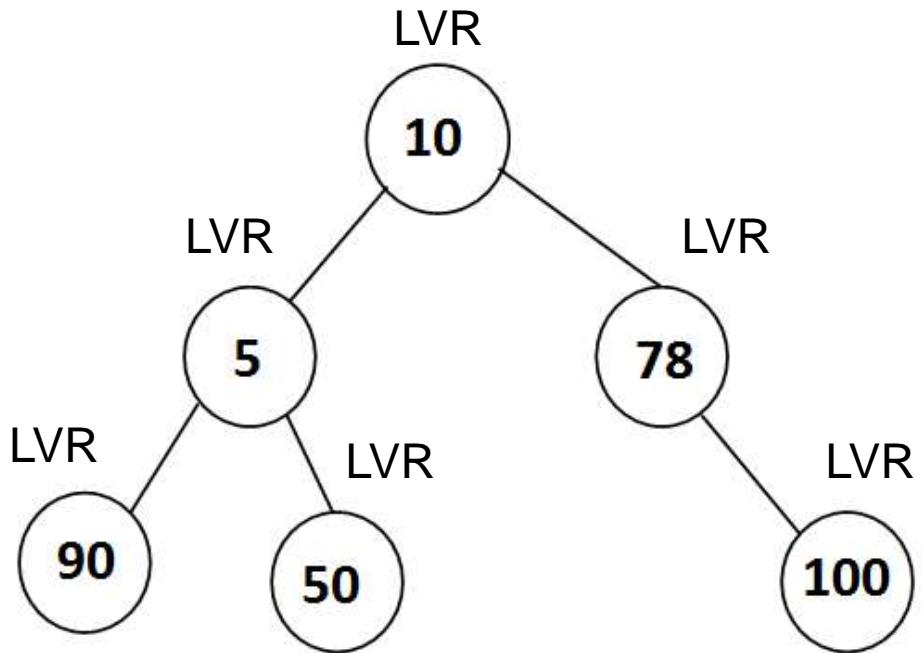
# In order Traversal (LVR)

- Traverse **Left Sub tree**
- **Visit the Node**
- Traverse **Right Sub tree**
- Example consider the below Tree



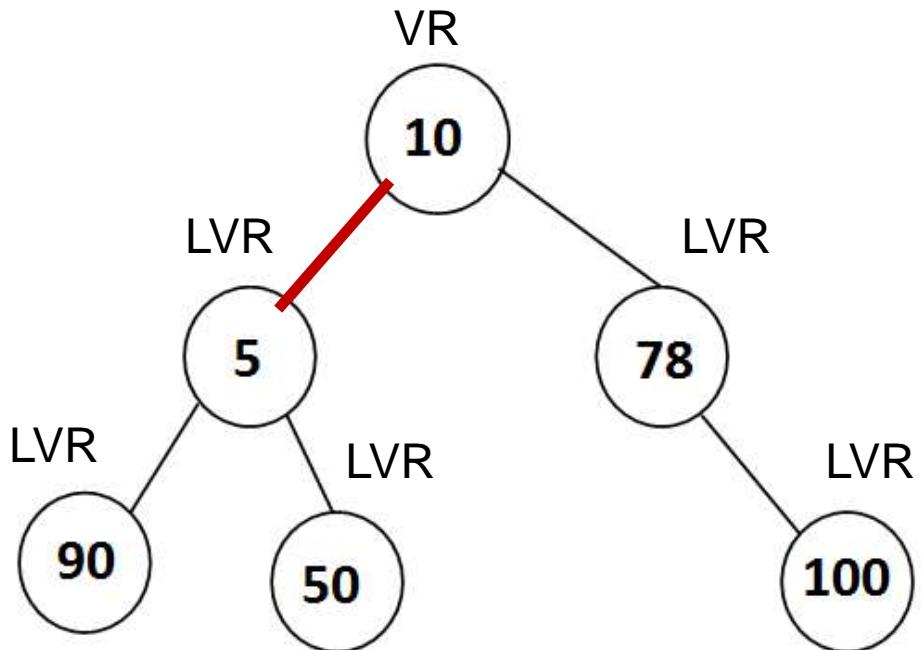


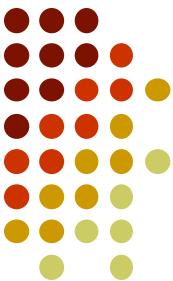
# In order Traversal (LVR)



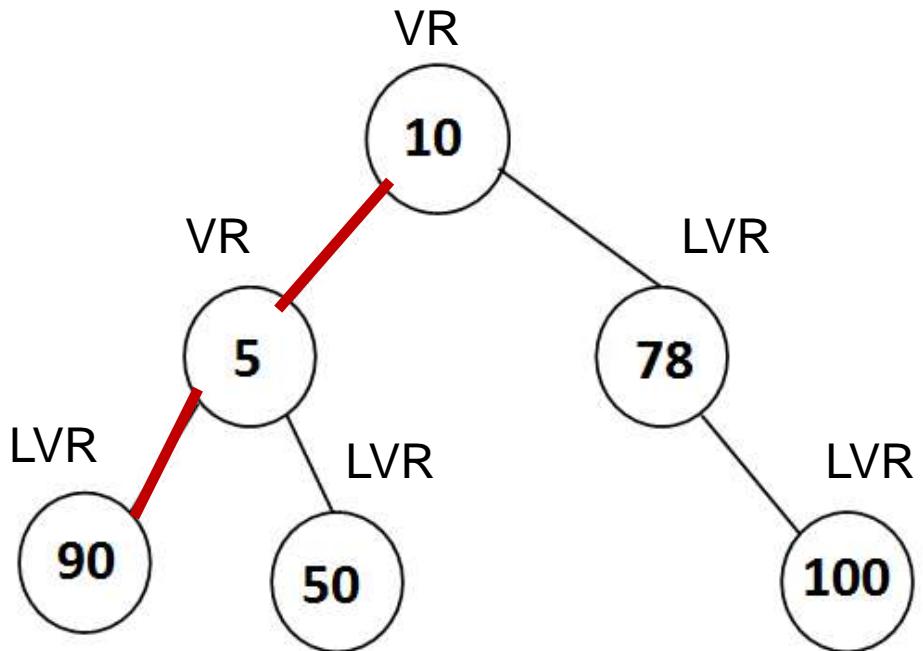


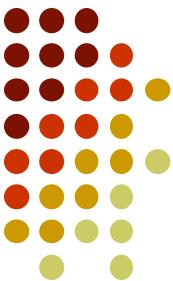
# In order Traversal (LVR)



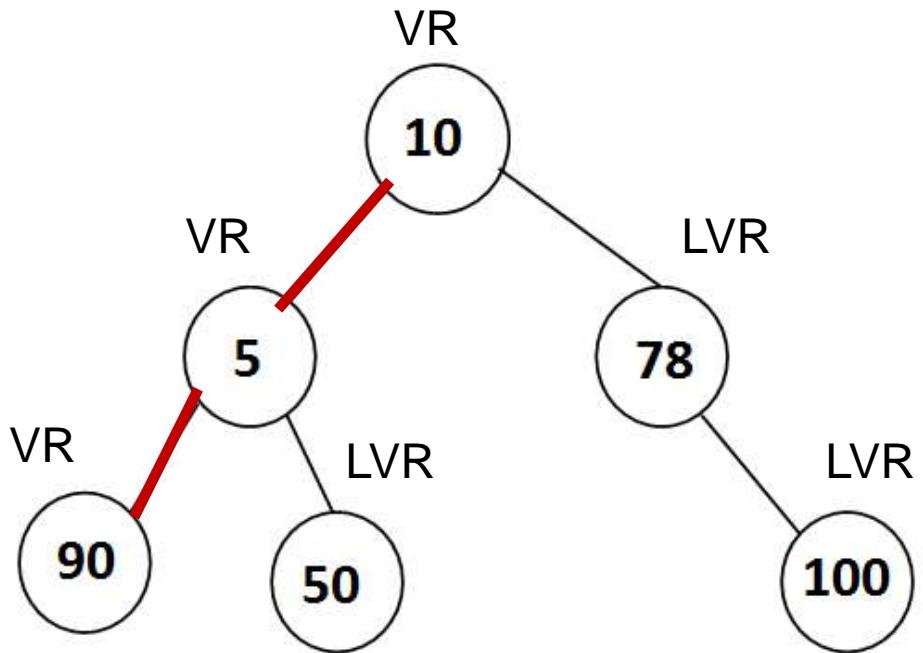


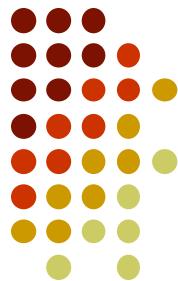
# In order Traversal (LVR)



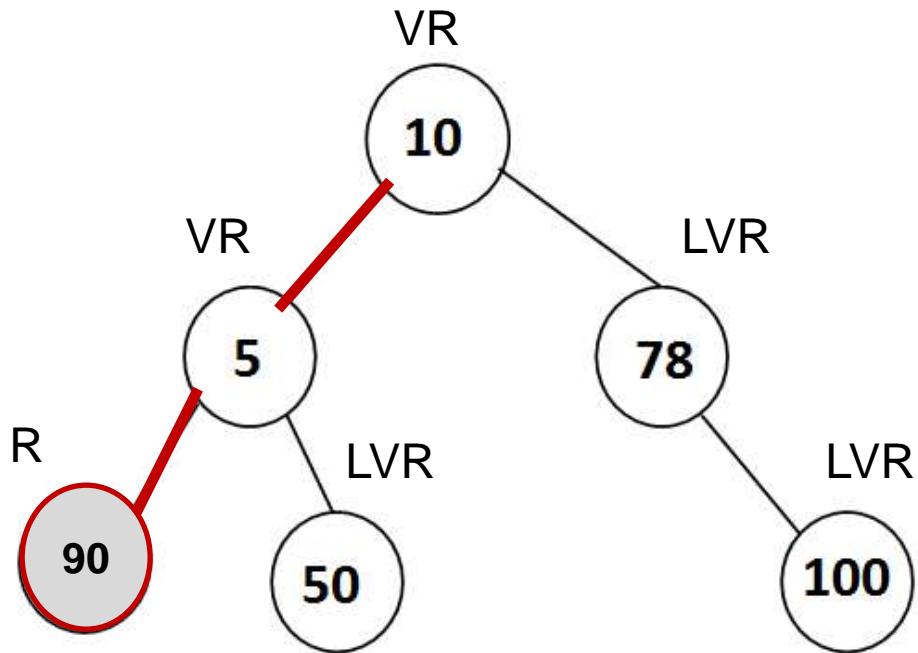


# In order Traversal (LVR)

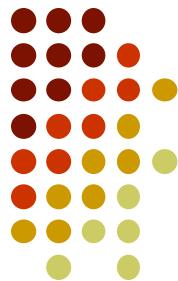




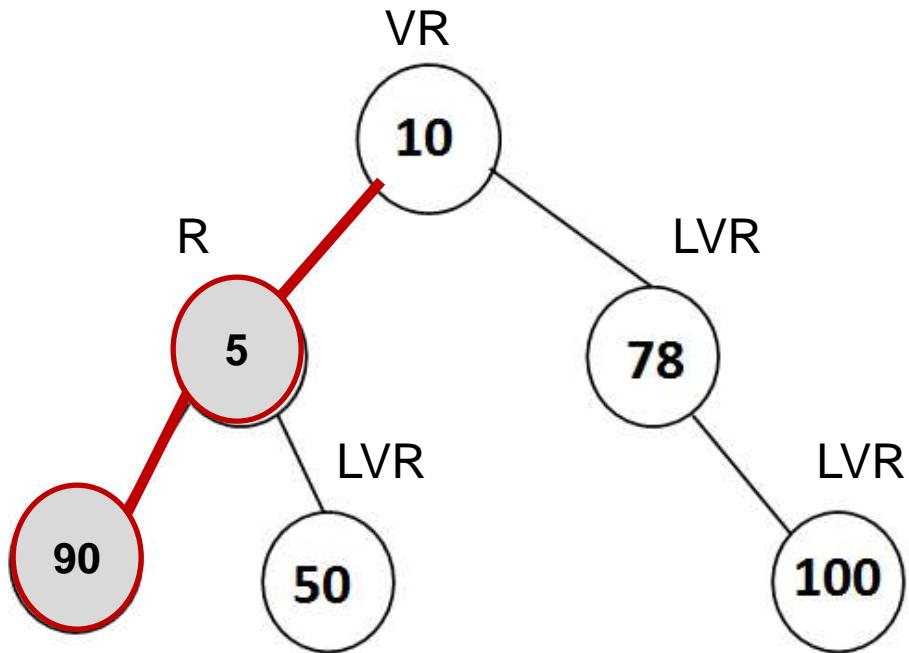
# In order Traversal (LVR)



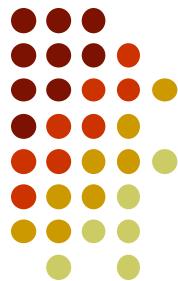
90



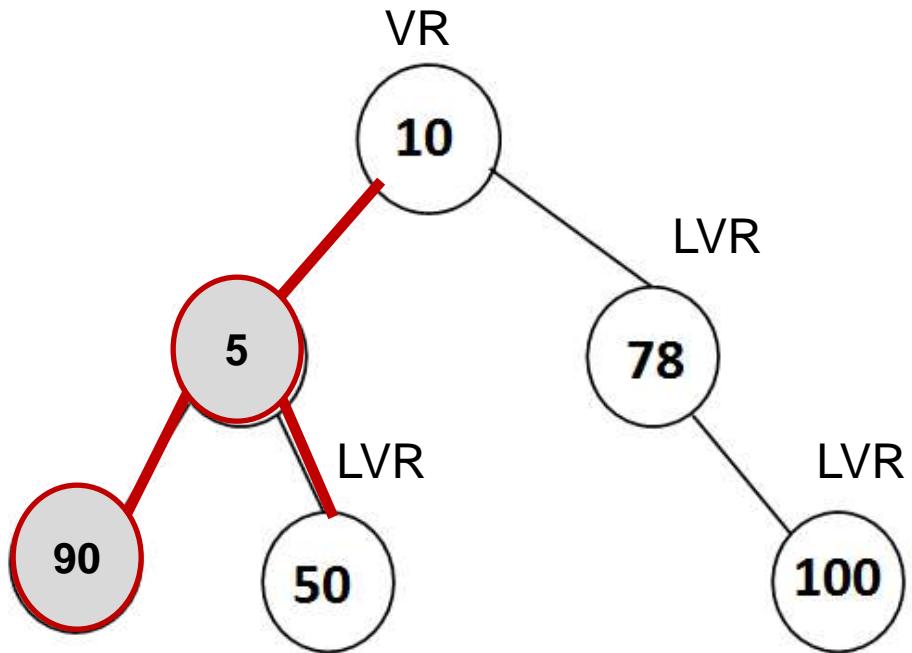
# In order Traversal (LVR)



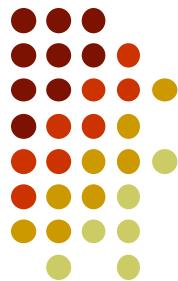
90 5



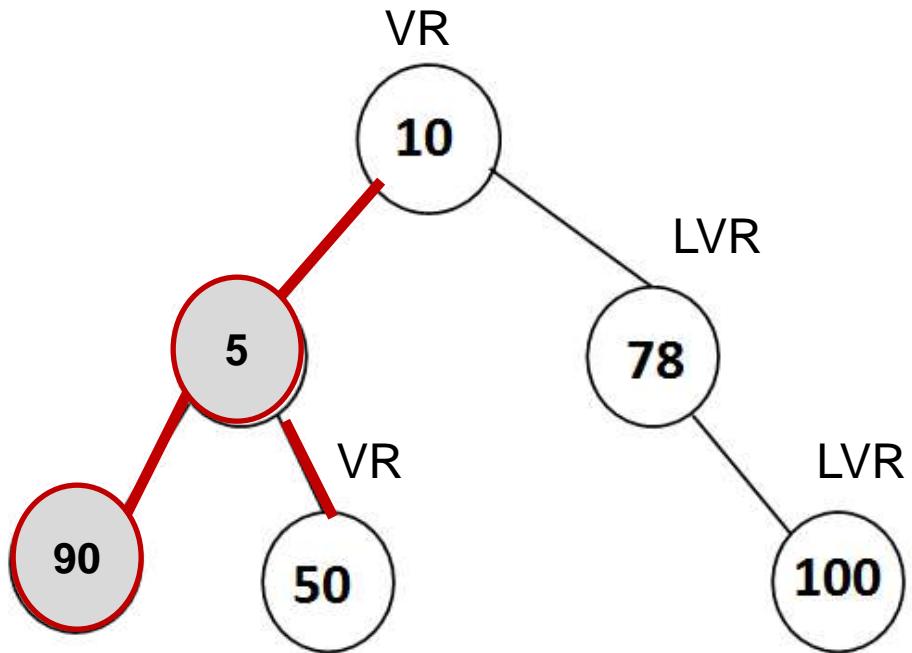
# In order Traversal (LVR)



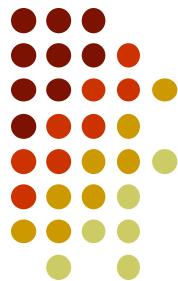
90 5



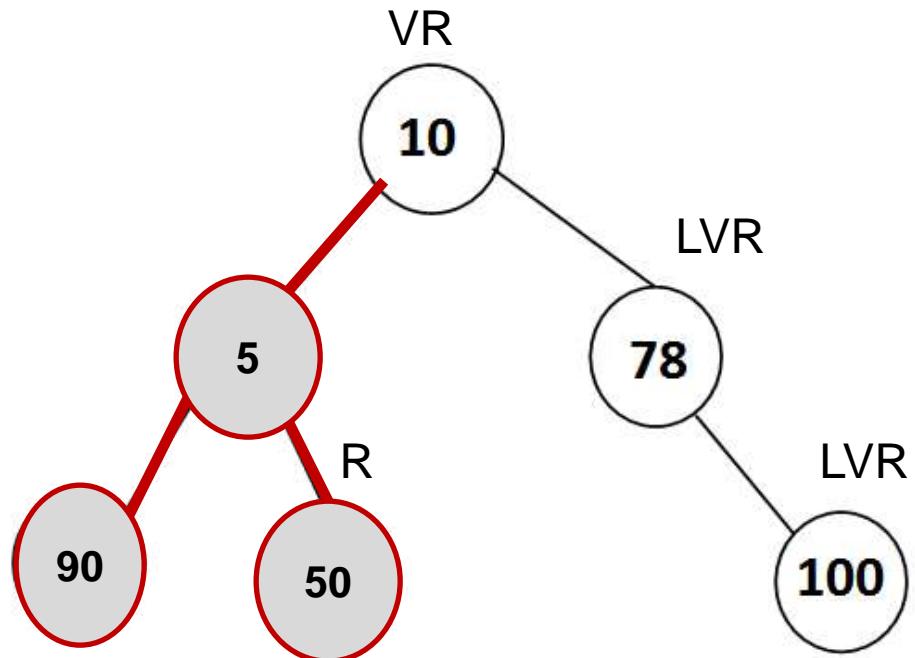
# In order Traversal (LVR)



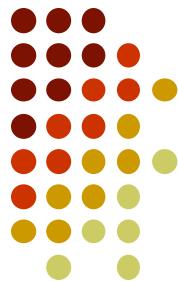
90 5



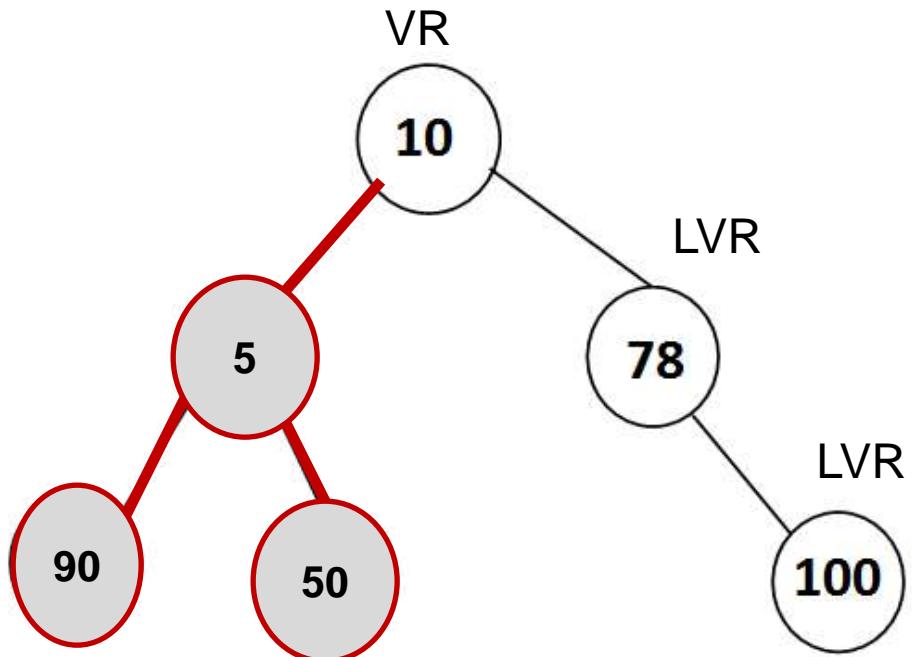
# In order Traversal (LVR)



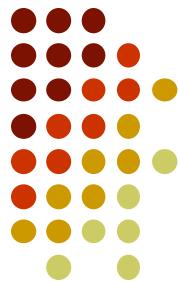
90 5 50



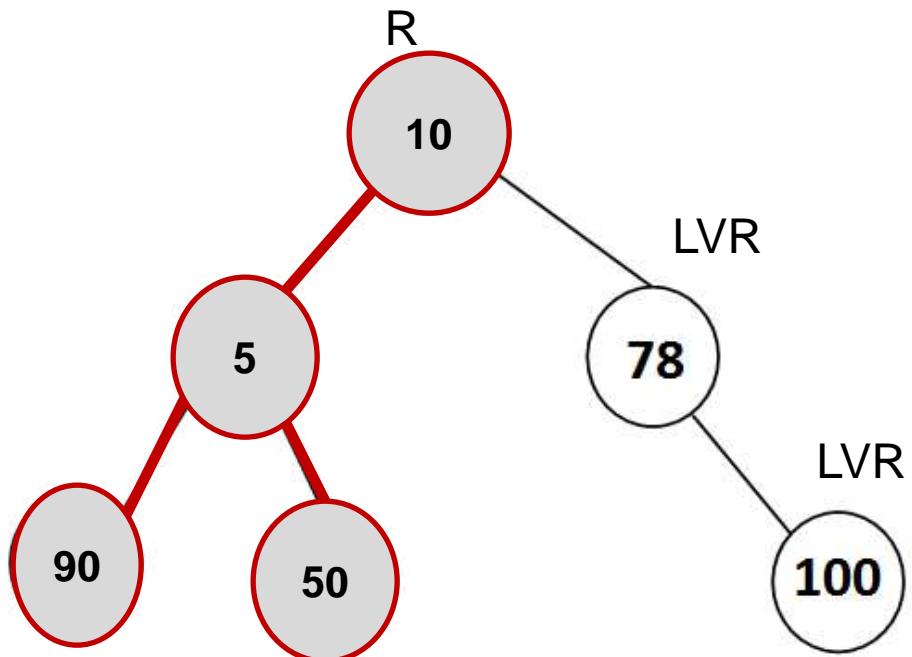
# In order Traversal (LVR)



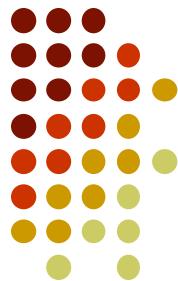
90 5 50



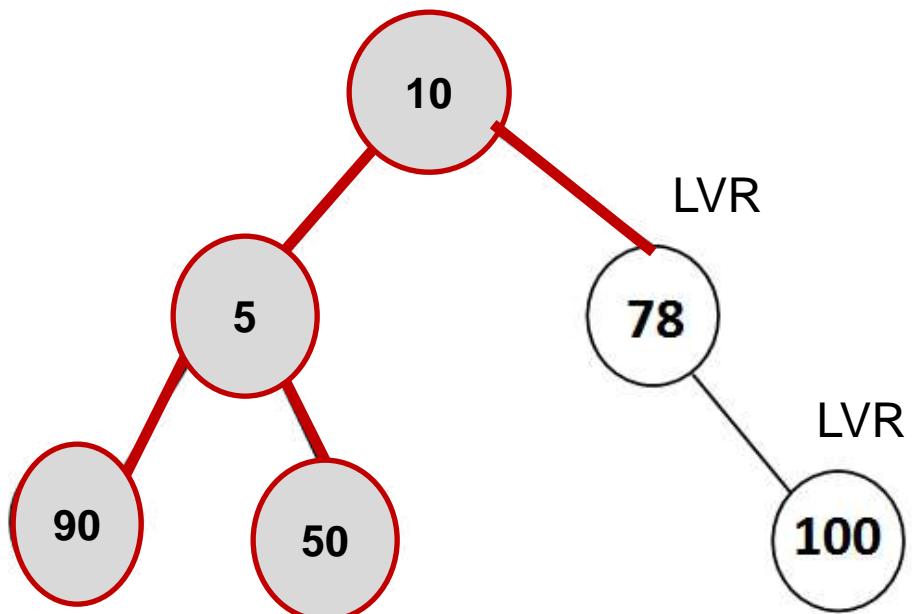
# In order Traversal (LVR)



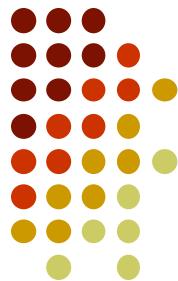
90 5 50 10



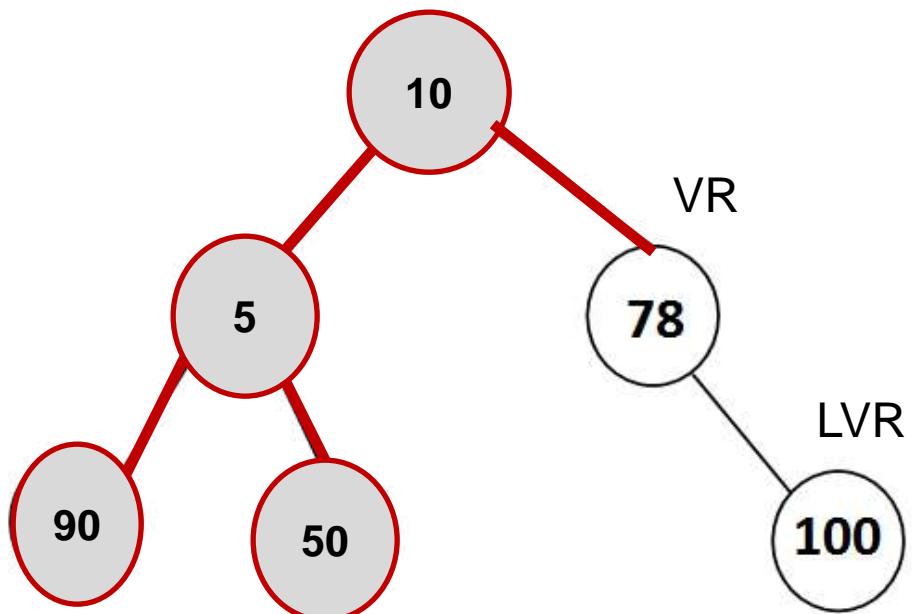
# In order Traversal (LVR)



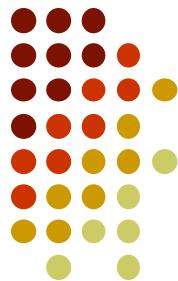
90 5 50 10



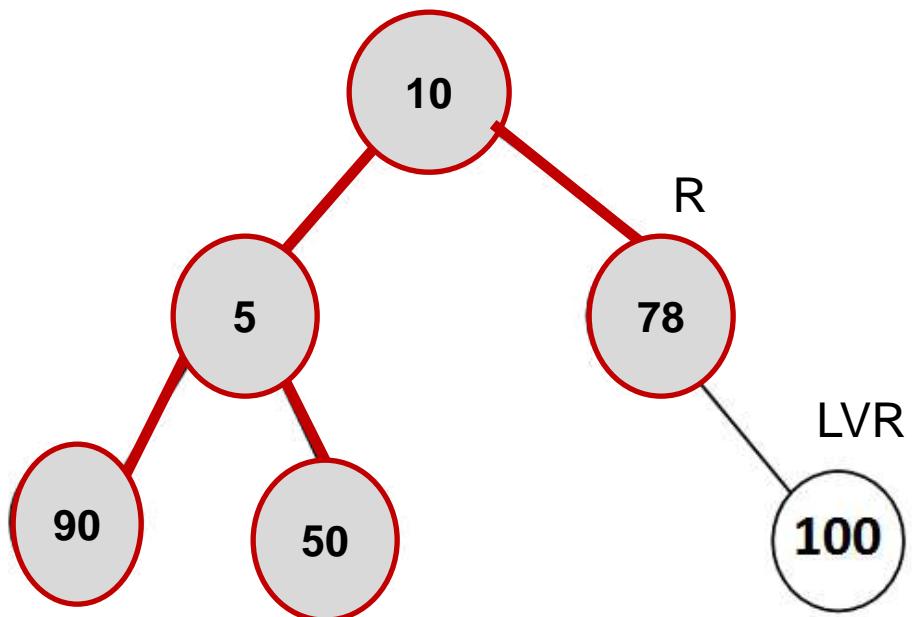
# In order Traversal (LVR)



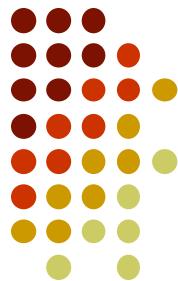
90 5 50 10



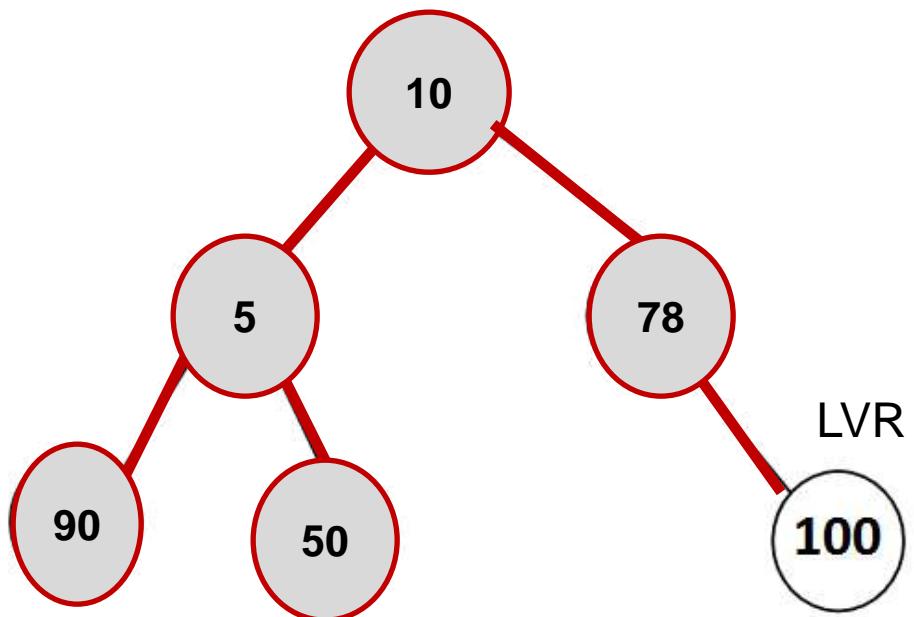
# In order Traversal (LVR)



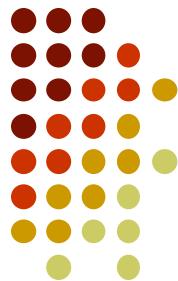
90 5 50 10 78



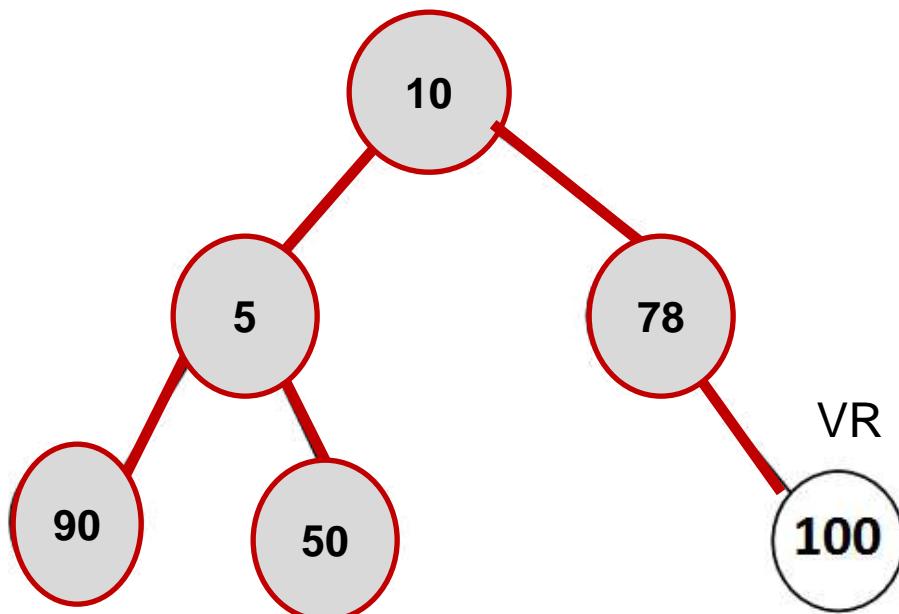
# In order Traversal (LVR)



90 5 50 10 78



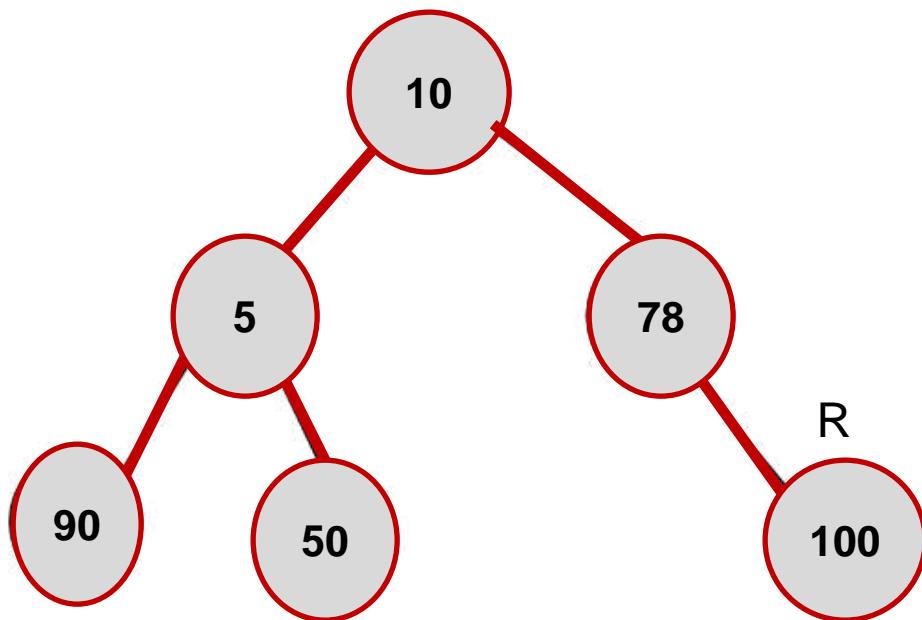
# In order Traversal (LVR)



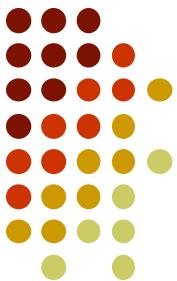
90 5 50 10 78



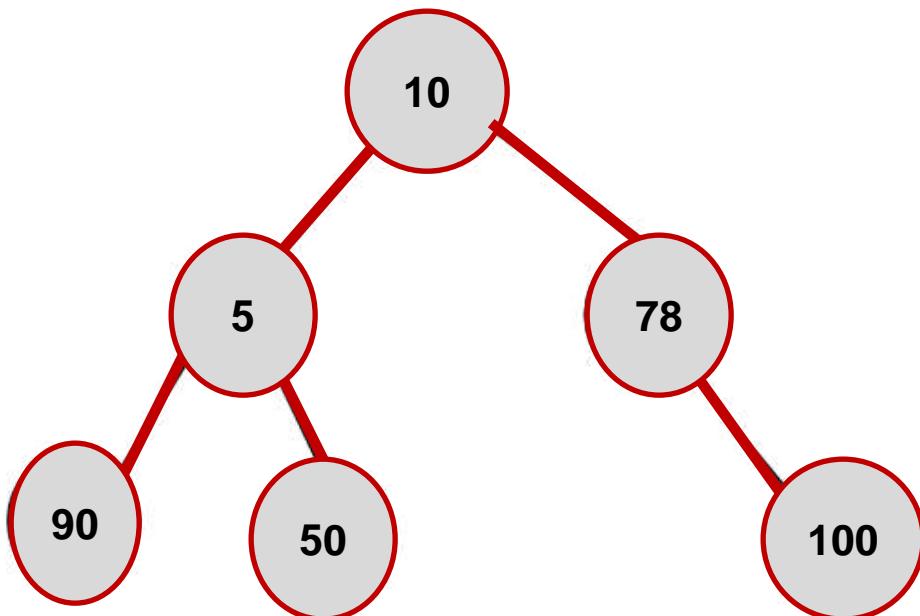
# In order Traversal (LVR)



90 5 50 10 78 100



# In order Traversal (LVR)



90 5 50 10 78 100

# In order Traversal Recursive function

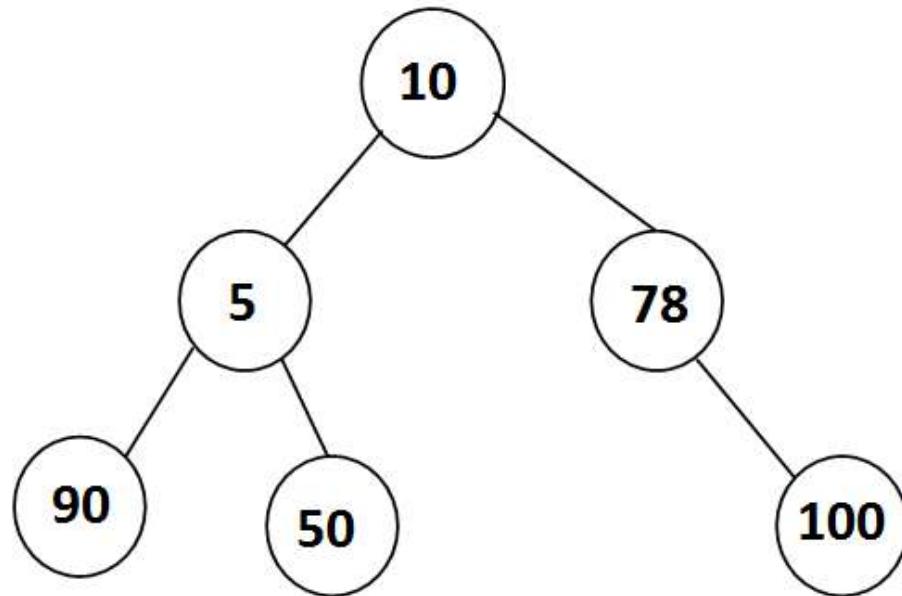


```
void in_order(root)
{
    if(root==NULL)
        return
    else
    {
        in_order(root->lchild);
        printf("%d",root->info);
        in_order(root->rchild);
    }
}
```



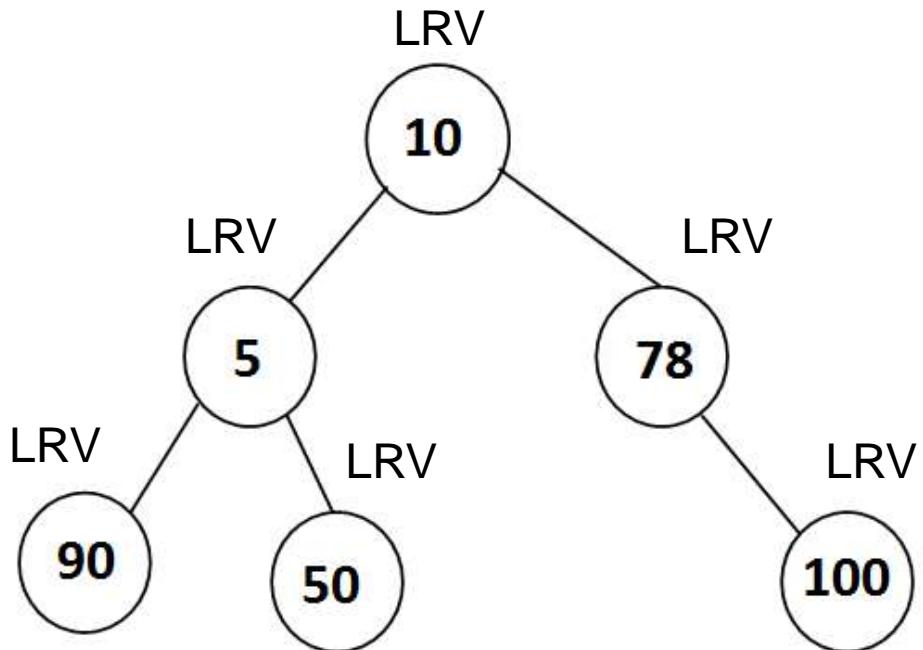
# Post order Traversal (LRV)

- Traverse **Left Sub tree**
- Traverse **Right Sub tree**
- **Visit the Node**
- Example consider the below Tree



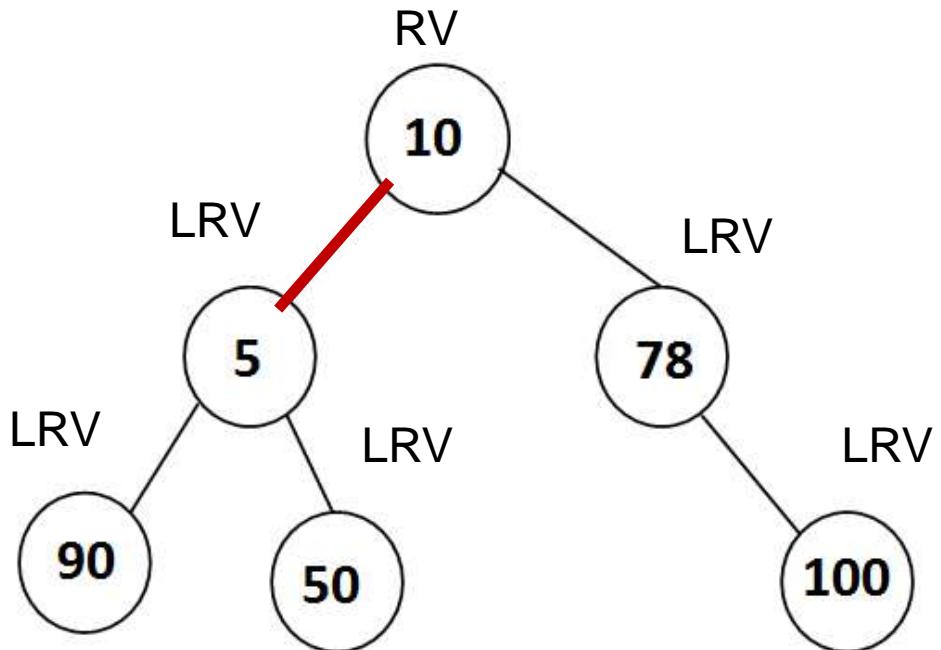


# Post order Traversal (LRV)



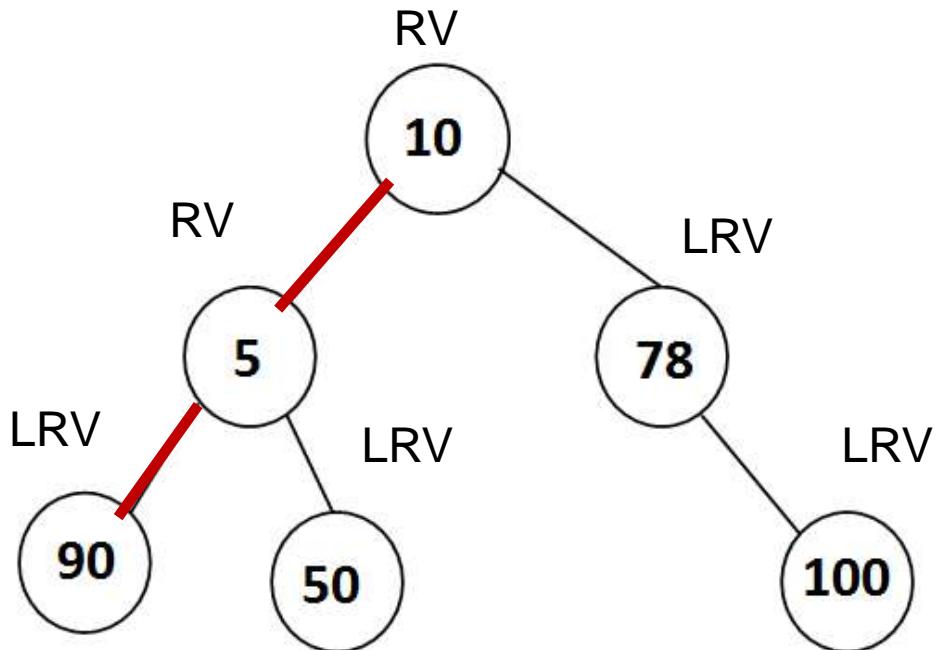


# Post order Traversal (LRV)



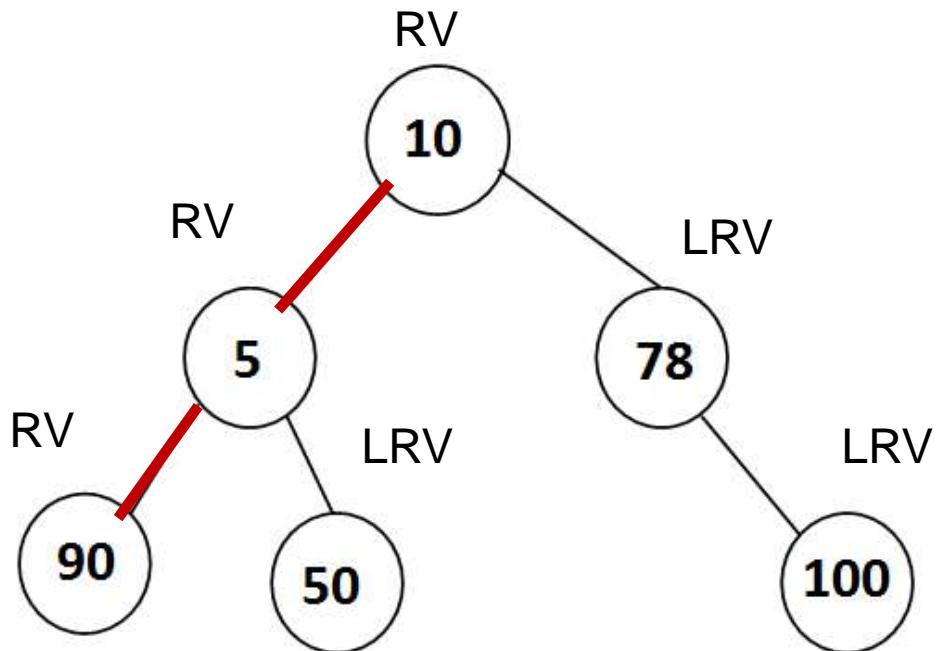


# Post order Traversal (LRV)



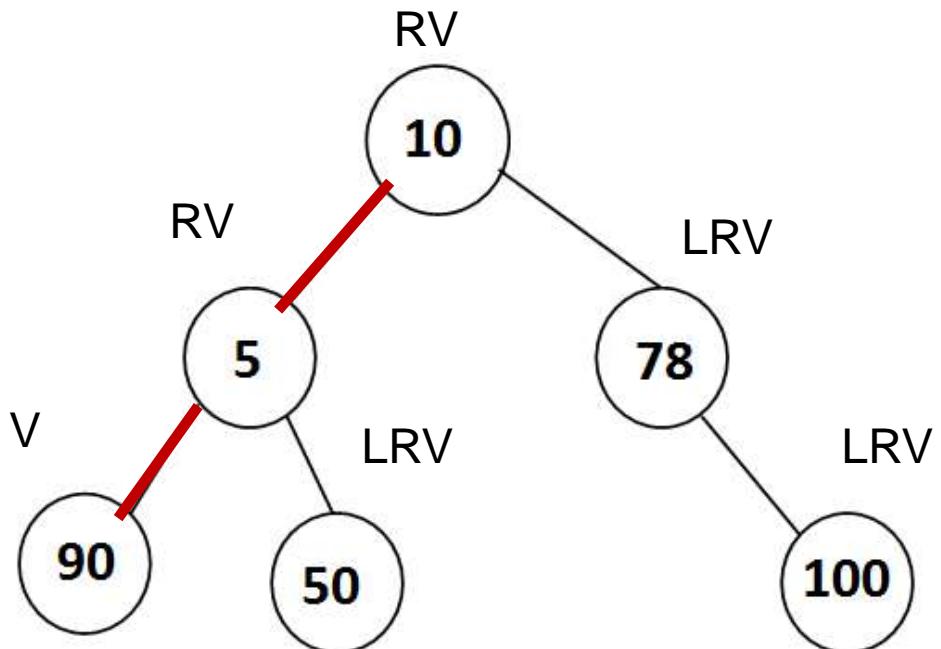


# Post order Traversal (LRV)



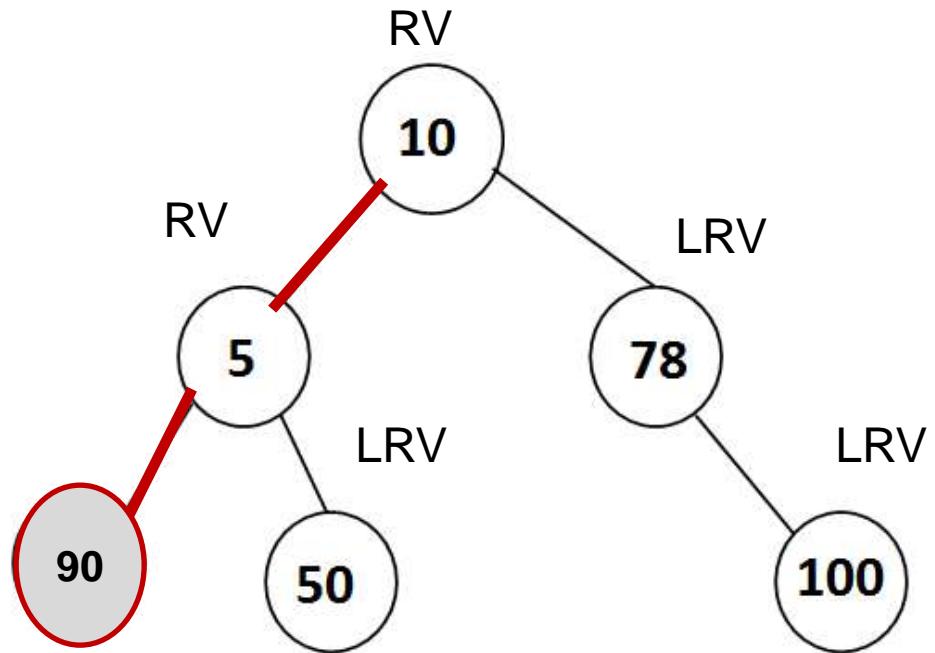


# Post order Traversal (LRV)

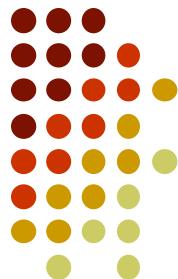




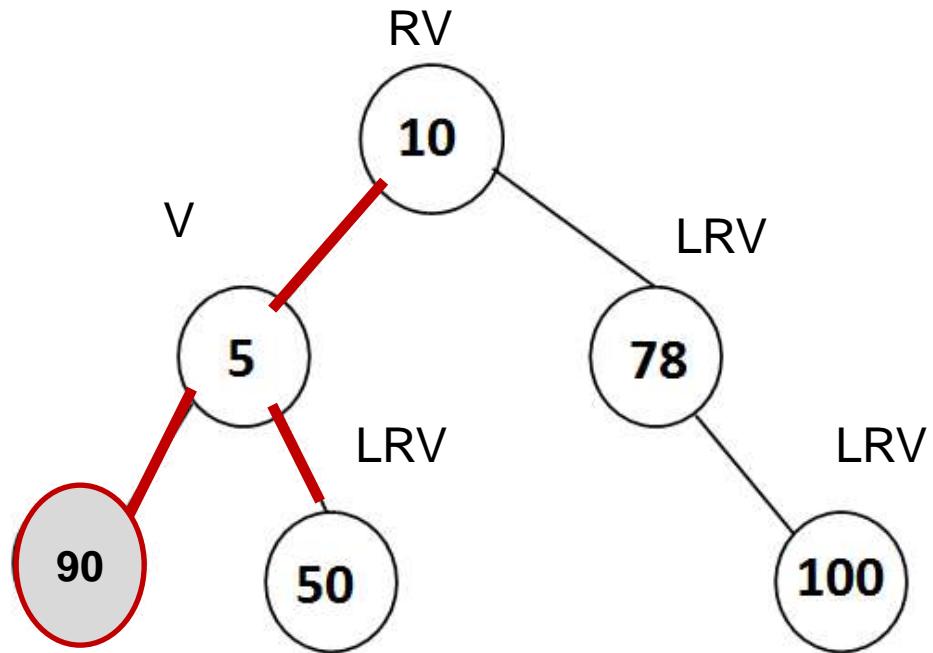
# Post order Traversal (LRV)



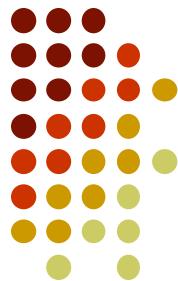
90



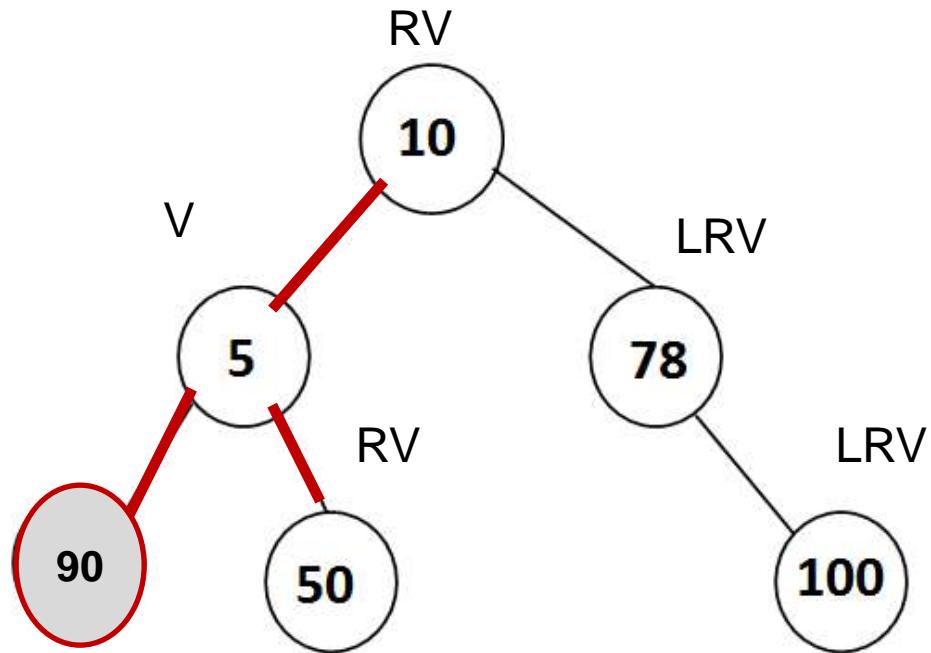
# Post order Traversal (LRV)



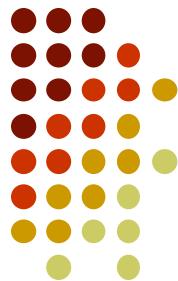
90



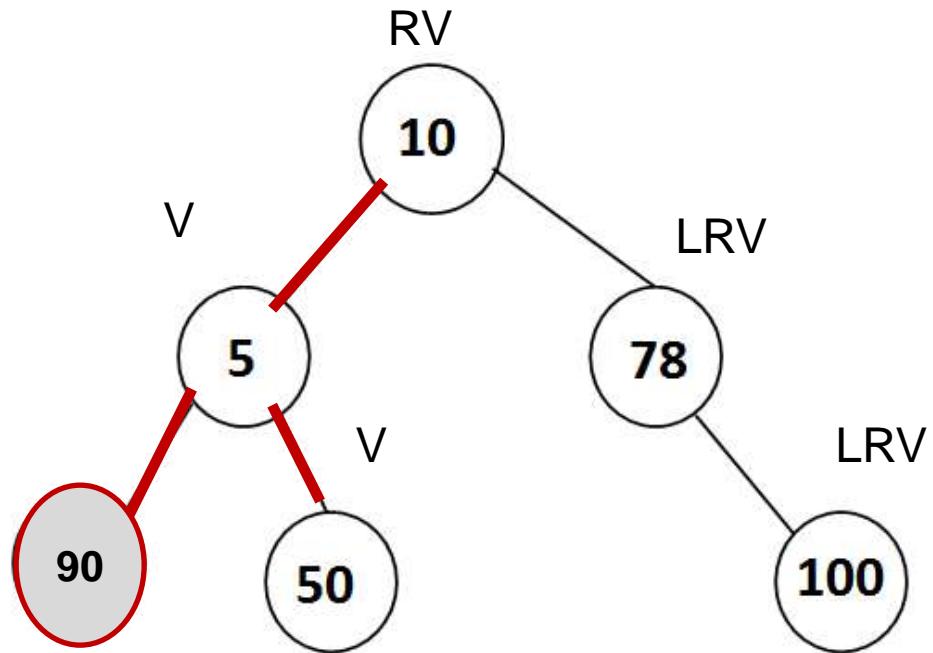
# Post order Traversal (LRV)



90



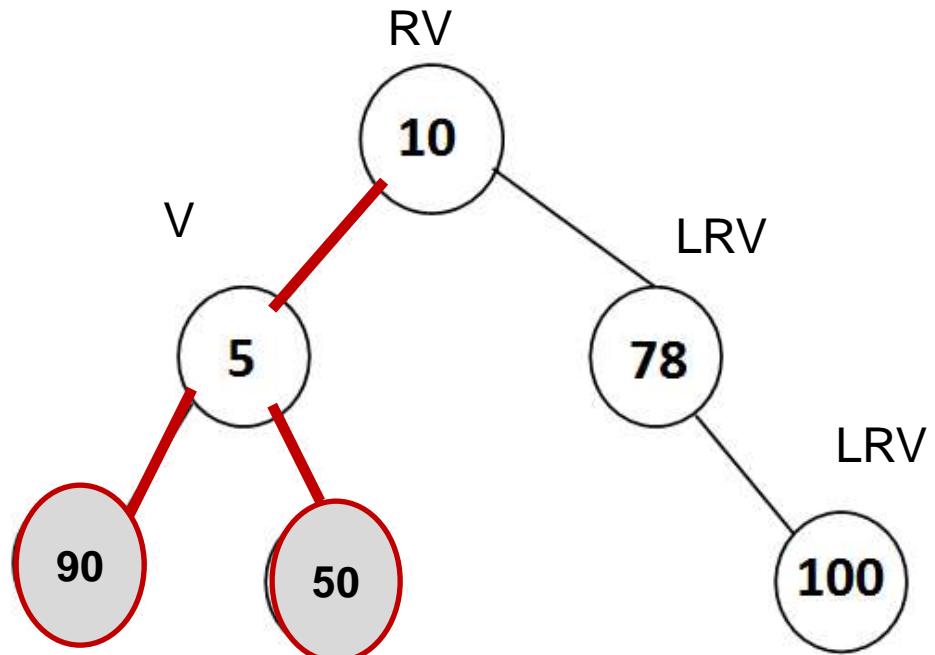
# Post order Traversal (LRV)



90



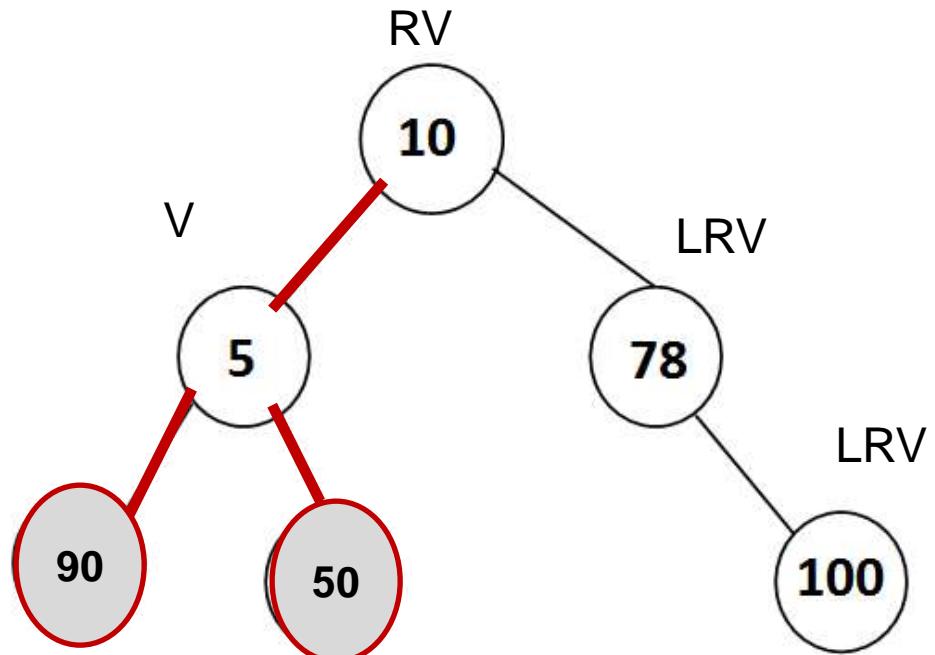
# Post order Traversal (LRV)



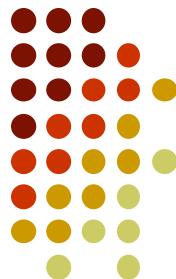
90 50



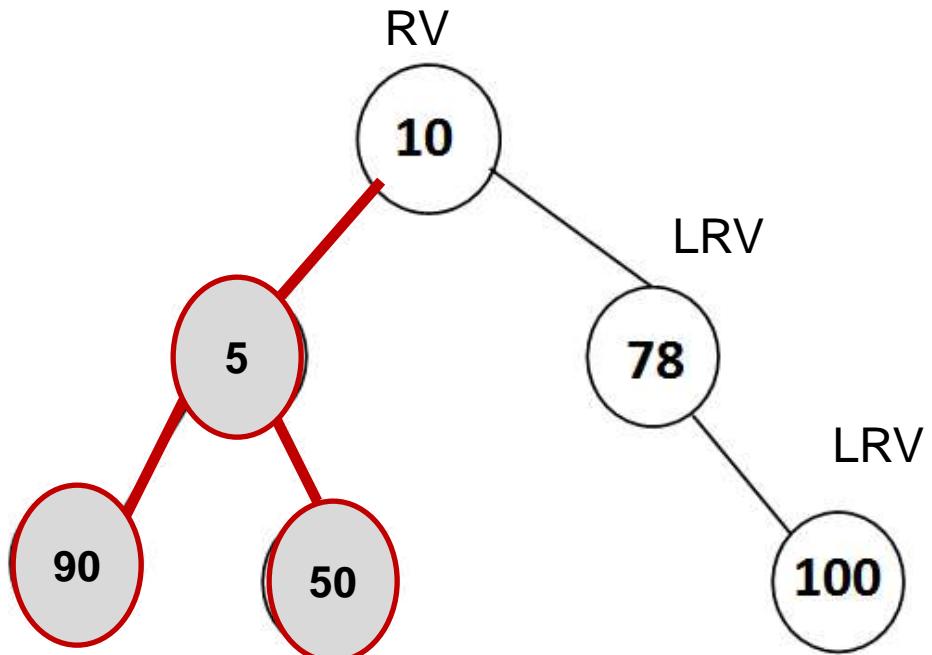
# Post order Traversal (LRV)



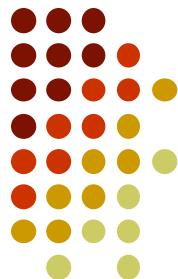
90 50



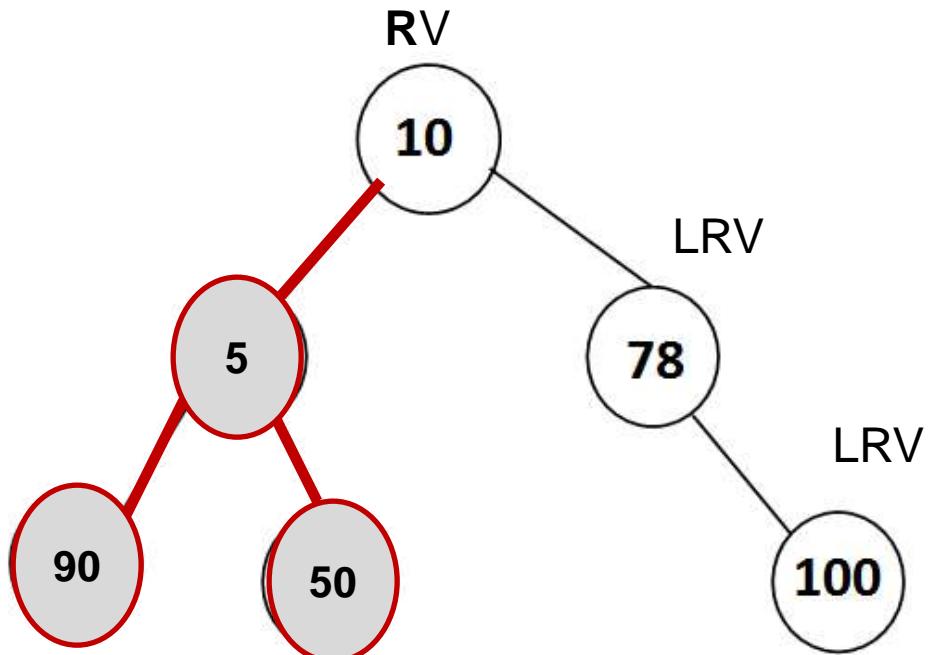
# Post order Traversal (LRV)



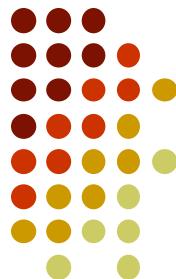
90 50 5



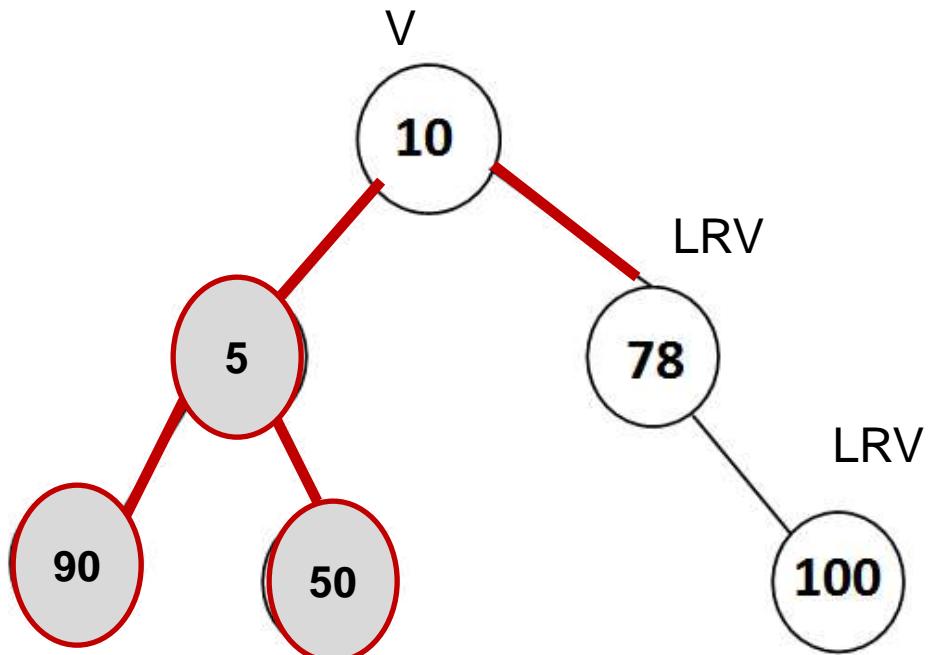
# Post order Traversal (LRV)



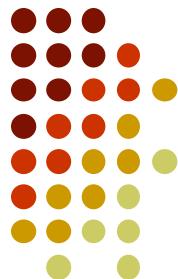
90 50 5



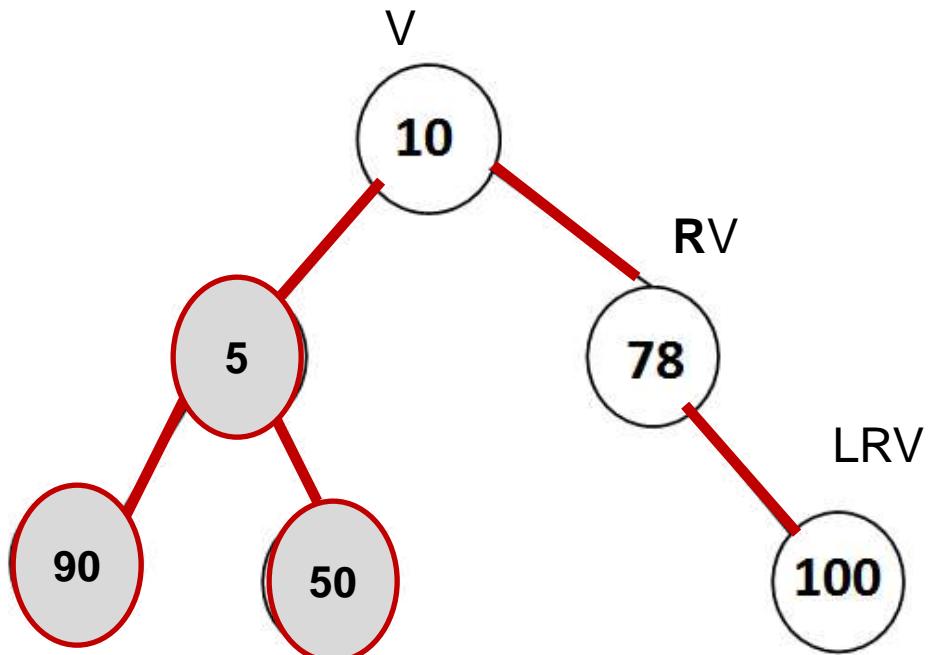
# Post order Traversal (LRV)



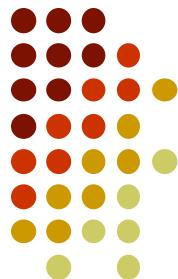
90 50 5



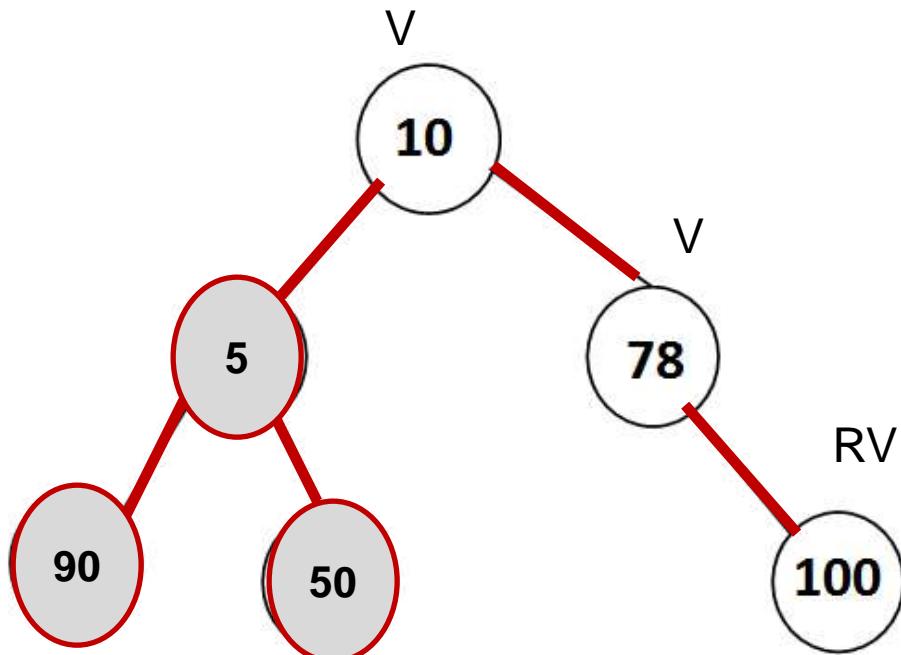
# Post order Traversal (LRV)



90 50 5



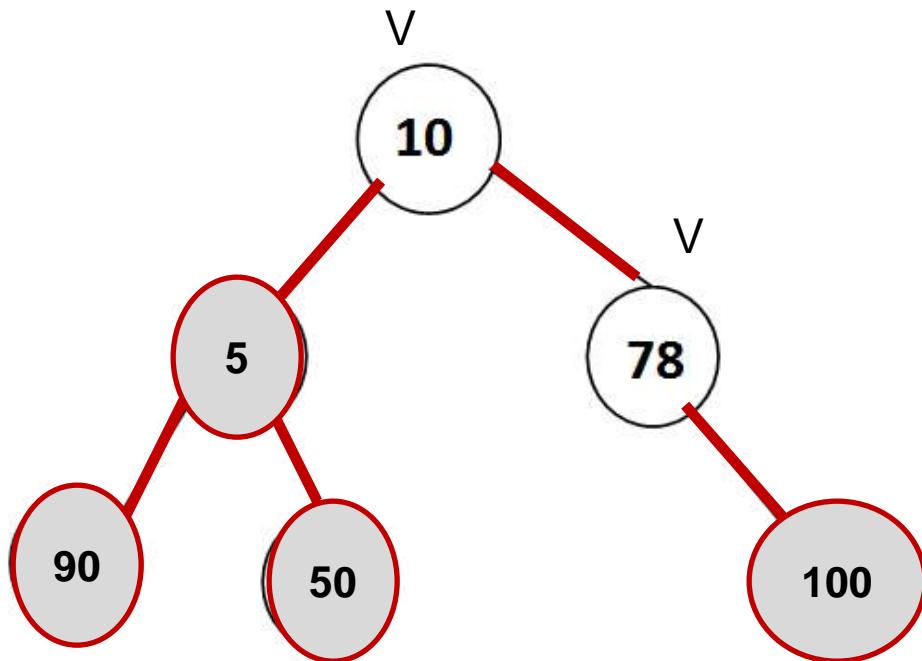
# Post order Traversal (LRV)



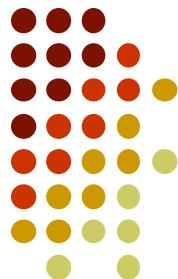
90 50 5



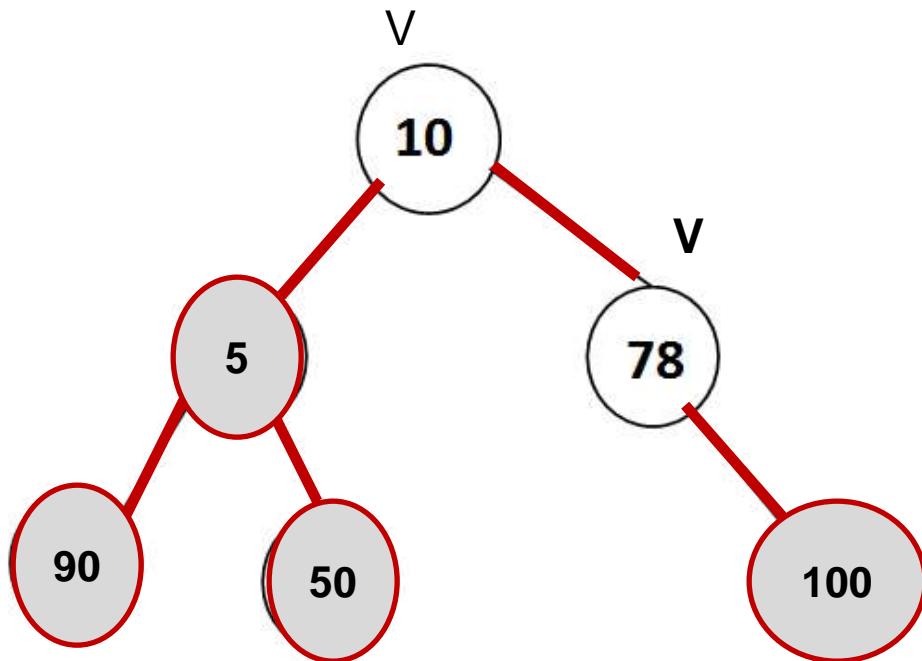
# Post order Traversal (LRV)



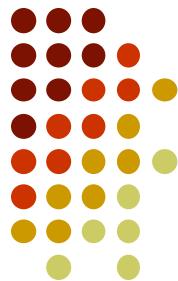
90 50 5 100



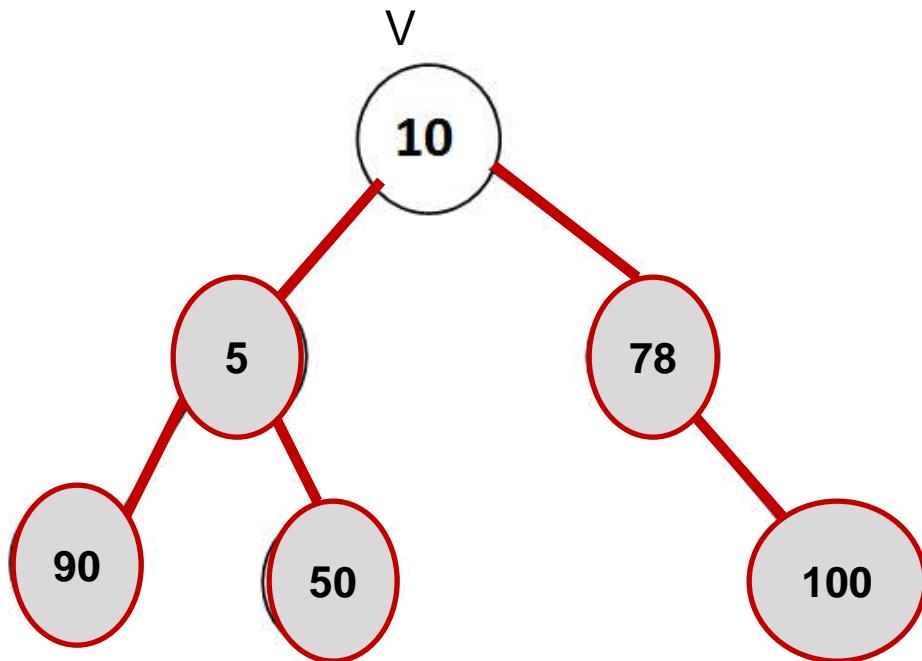
# Post order Traversal (LRV)



90 50 5 100



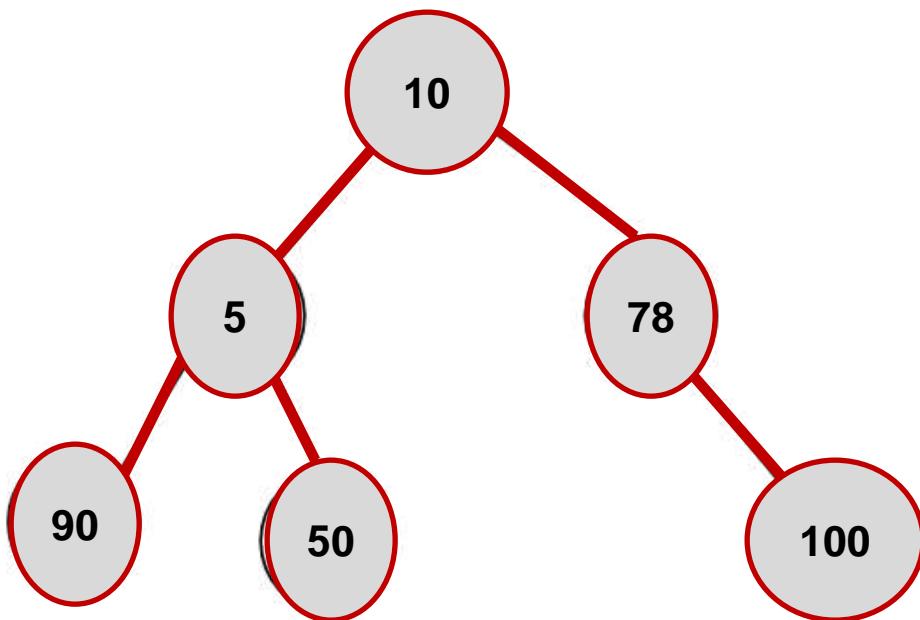
# Post order Traversal (LRV)



90 50 5 100 78



# Post order Traversal (LRV)



90 50 5 100 78 10

# Post order Traversal Recursive function

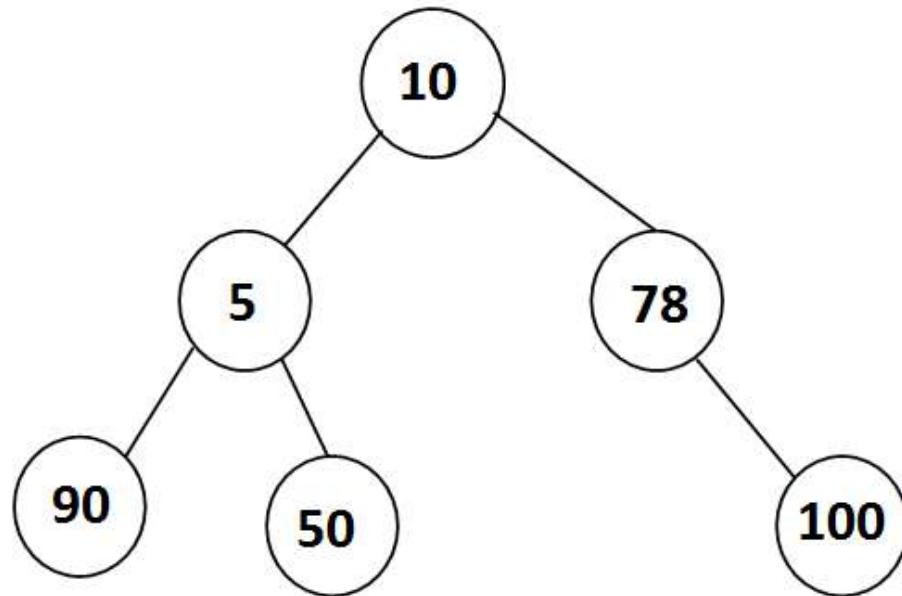


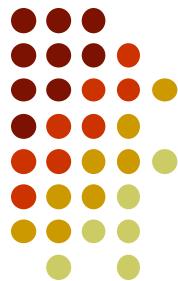
```
void post_order(root)
{
    if(root==NULL)
        return
    else
    {
        post_order(root->lchild);
        post_order(root->rchild);
        printf("%d",root->info);
    }
}
```



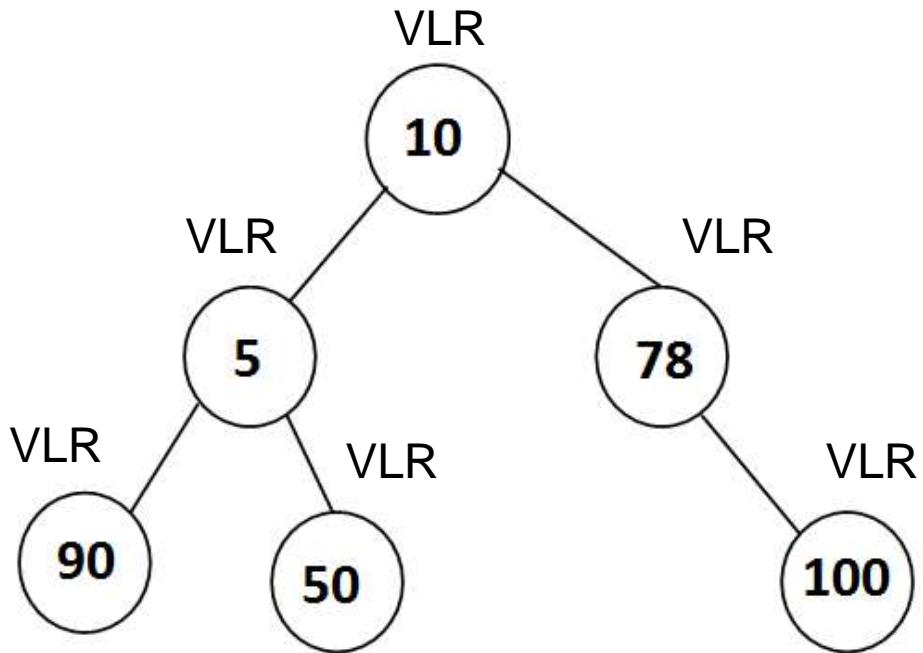
# Pre order Traversal (VLR)

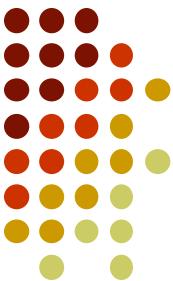
- Visit the Node
- Traverse Left Sub tree
- Traverse Right Sub tree
- Example consider the below Tree



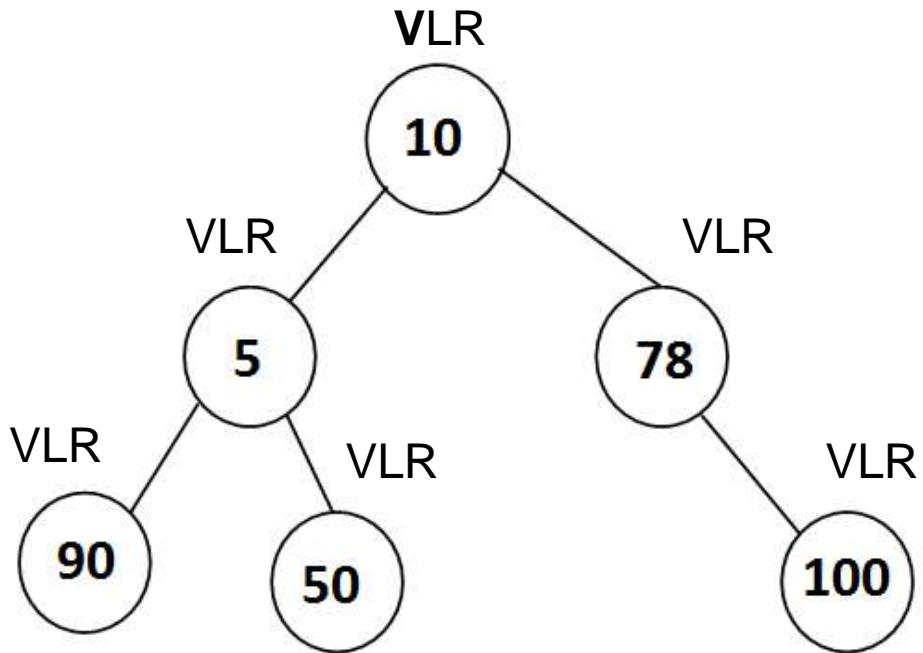


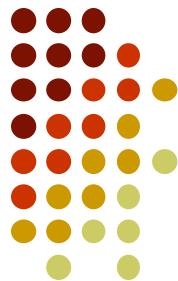
# In order Traversal (LVR)



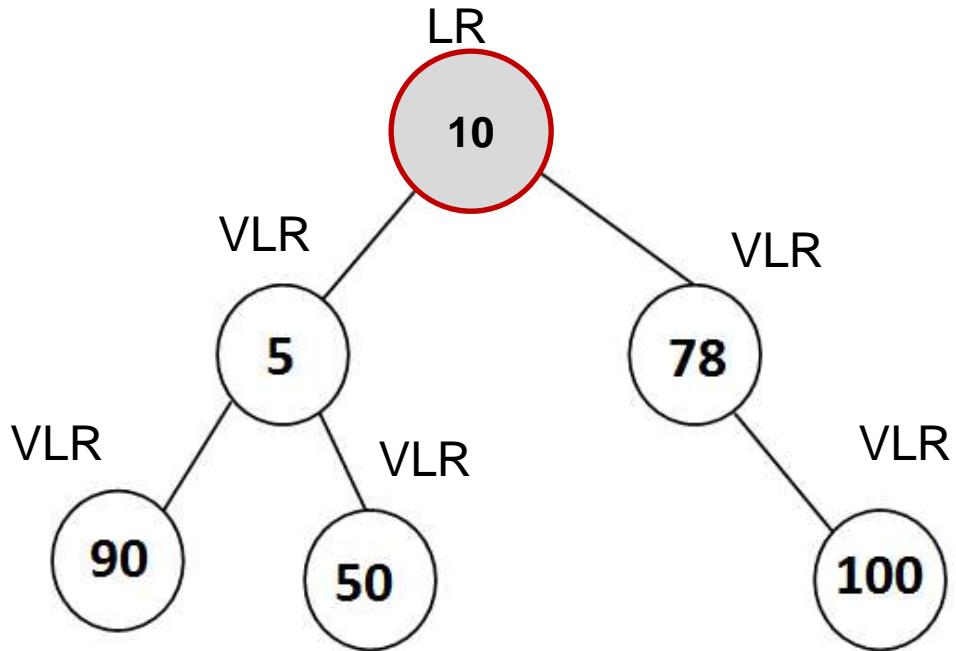


# In order Traversal (LVR)

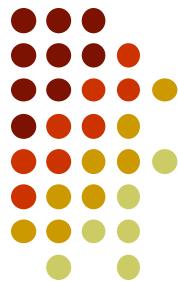




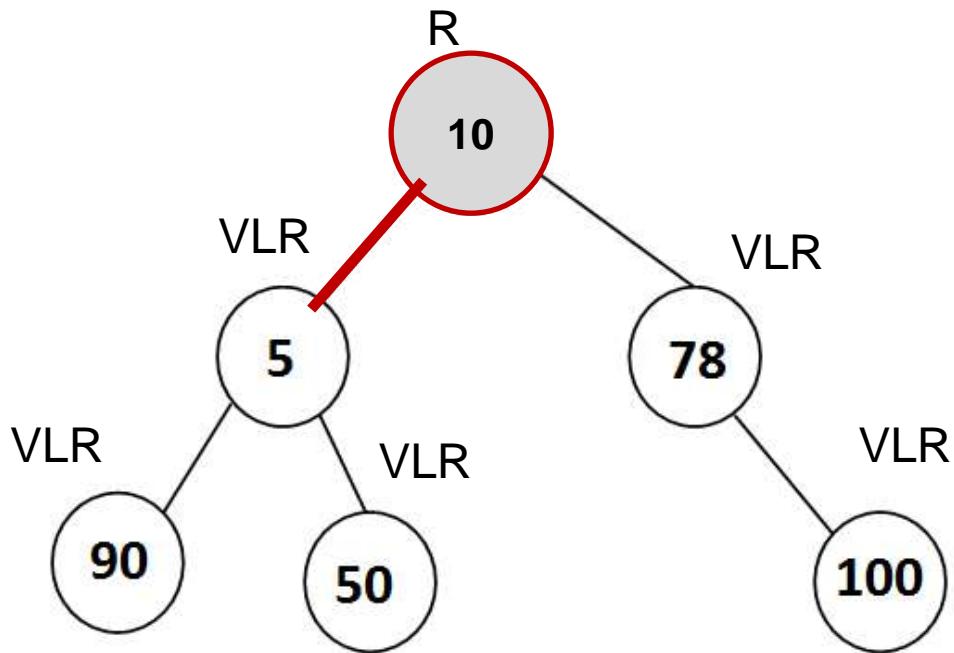
# Pre order Traversal (VLR)

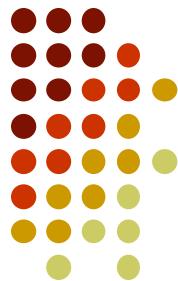


10

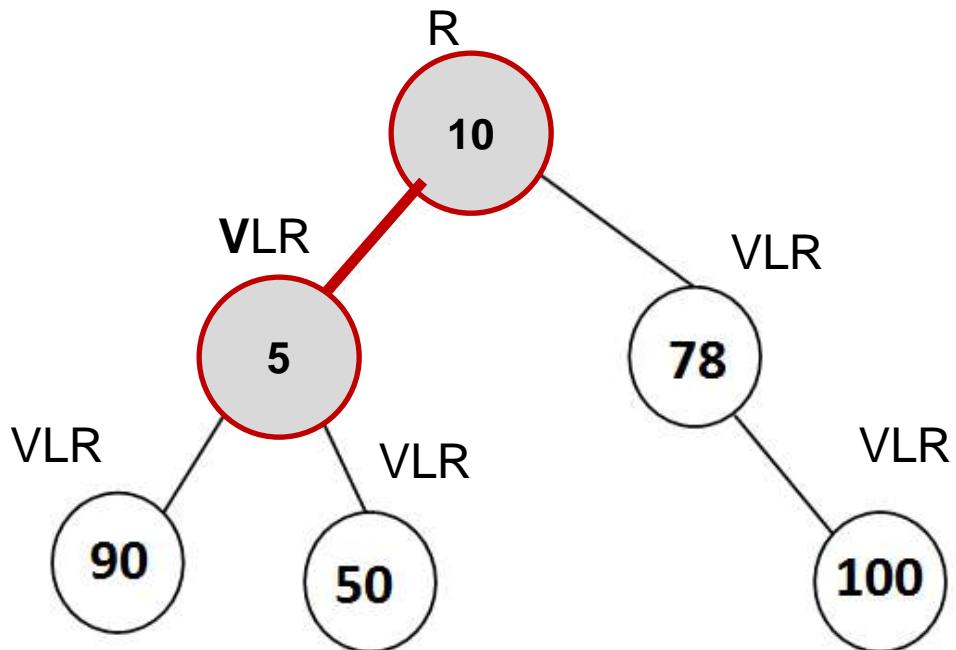


# Pre order Traversal (VLR)

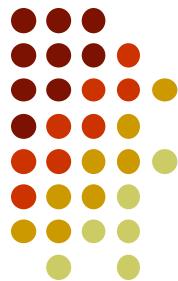




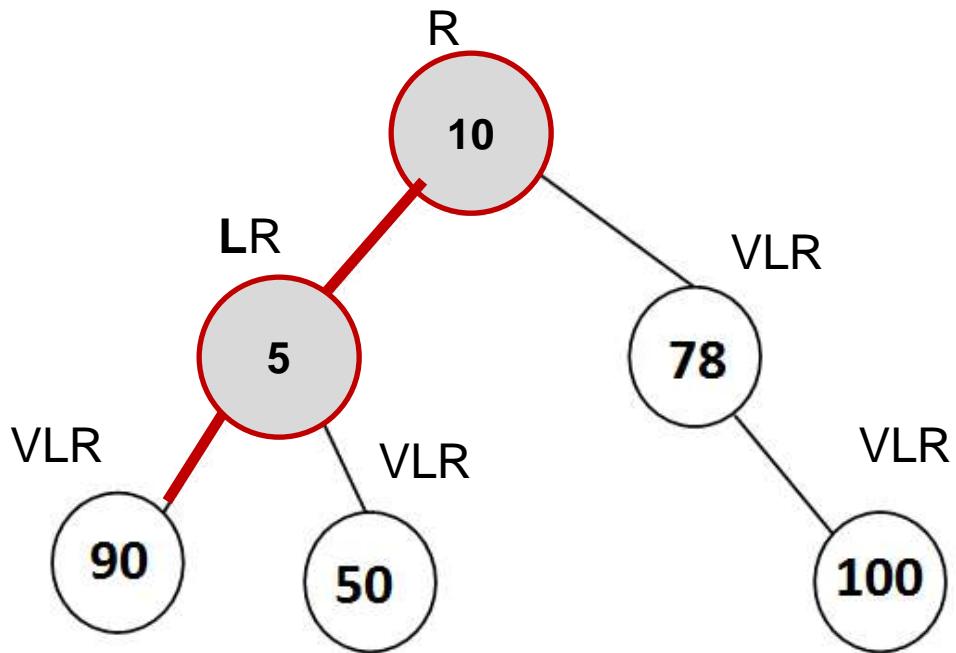
# Pre order Traversal (VLR)



10 5



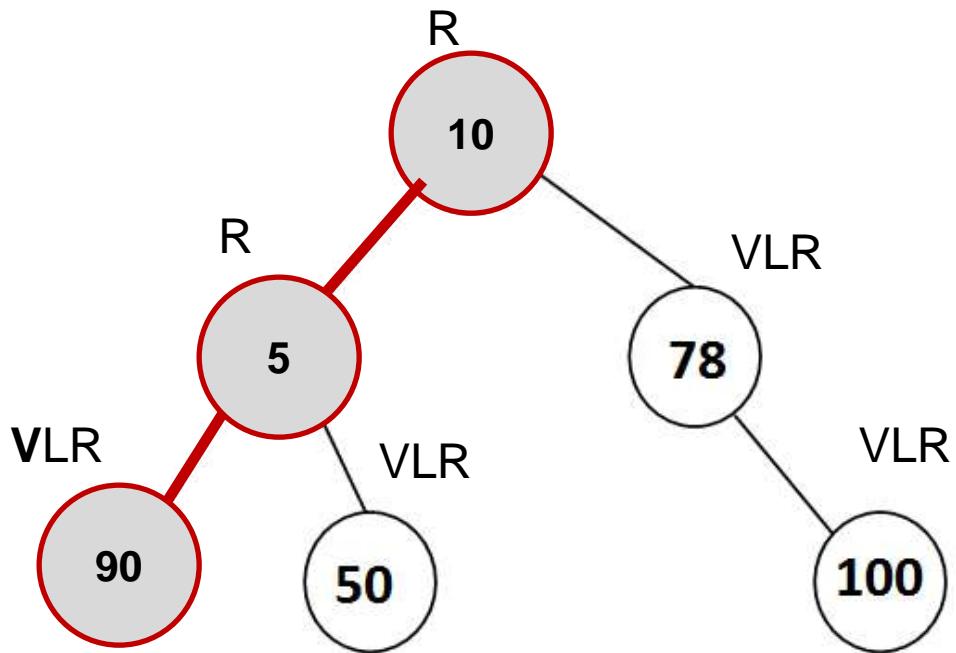
# Pre order Traversal (VLR)



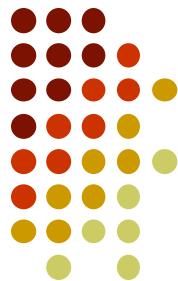
10 5



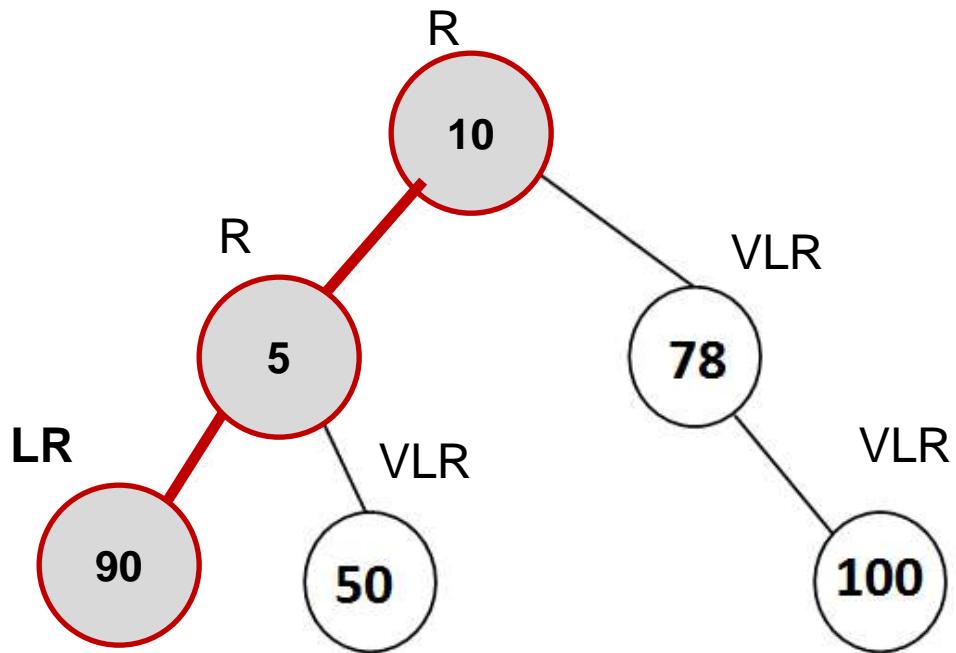
# Pre order Traversal (VLR)



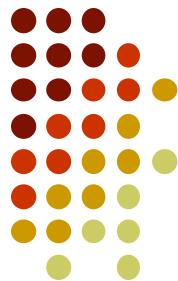
10 5 90



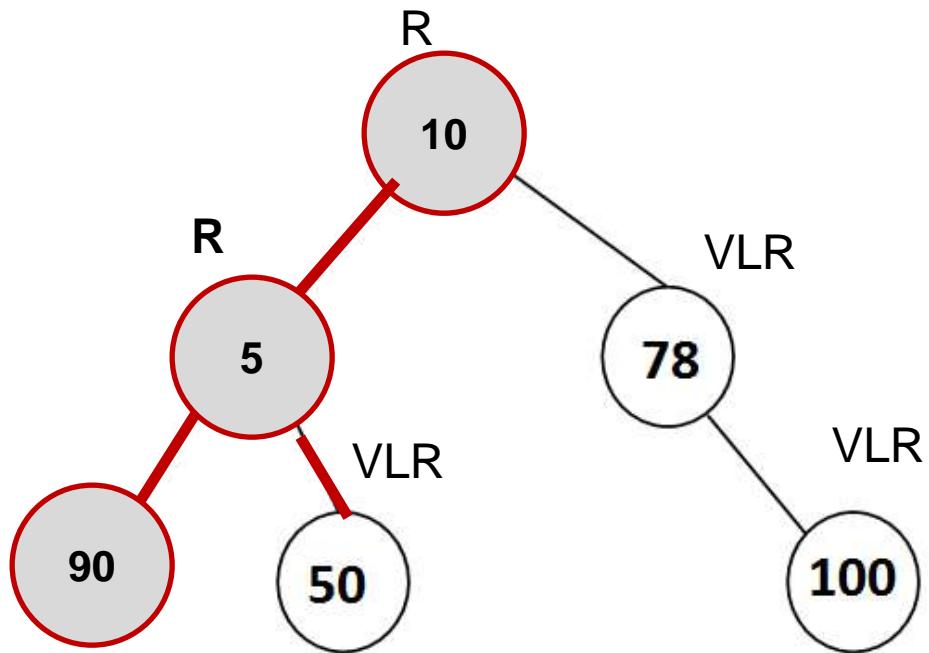
# Pre order Traversal (VLR)



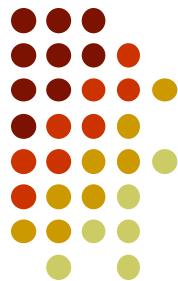
10 5 90



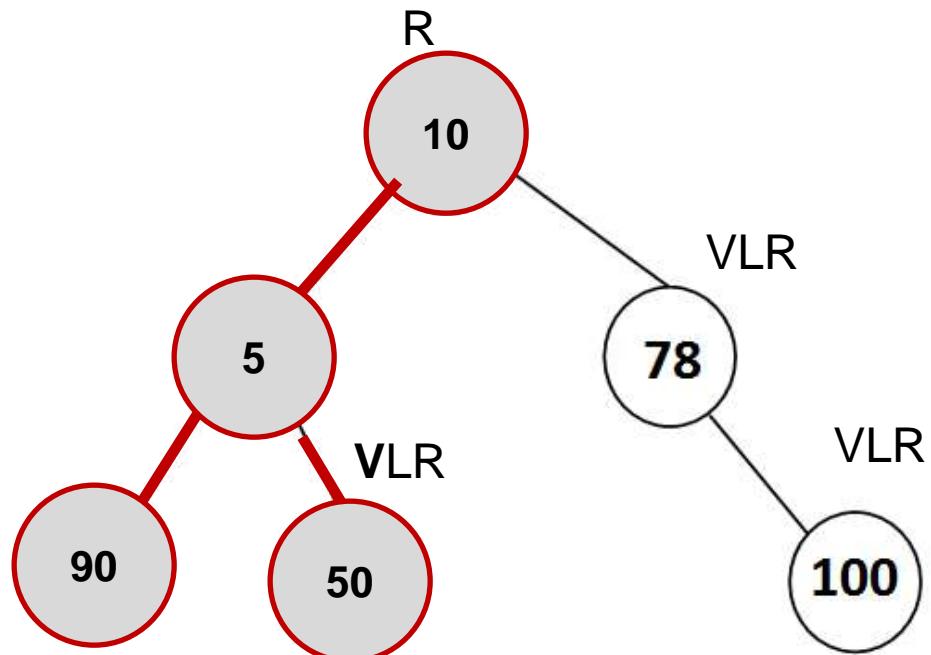
# Pre order Traversal (VLR)



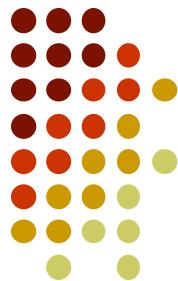
10 5 90



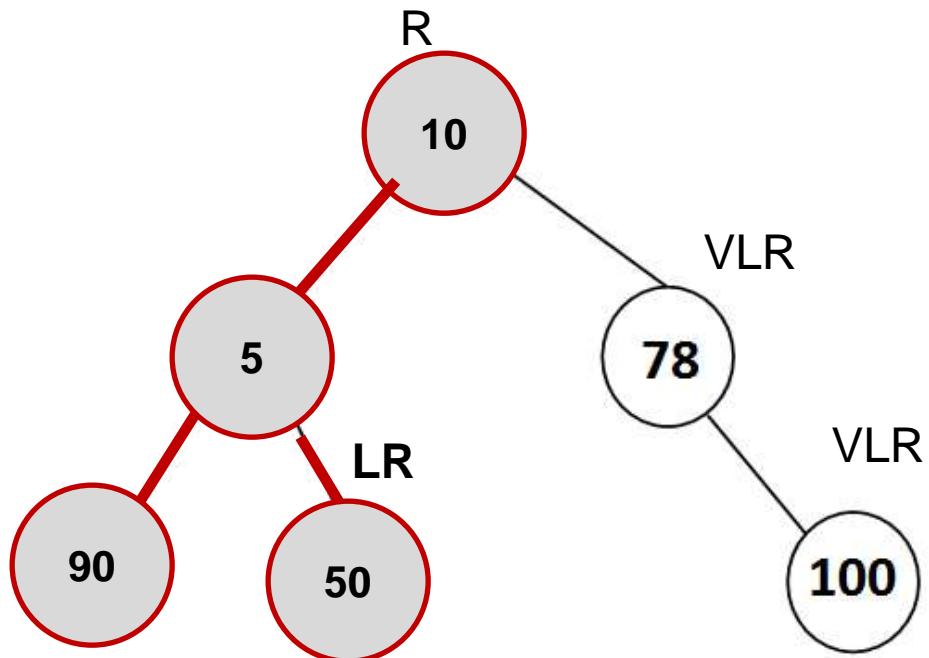
# Pre order Traversal (VLR)



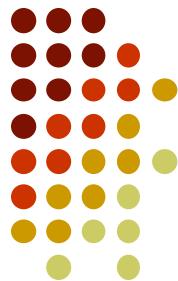
10 5 90 50



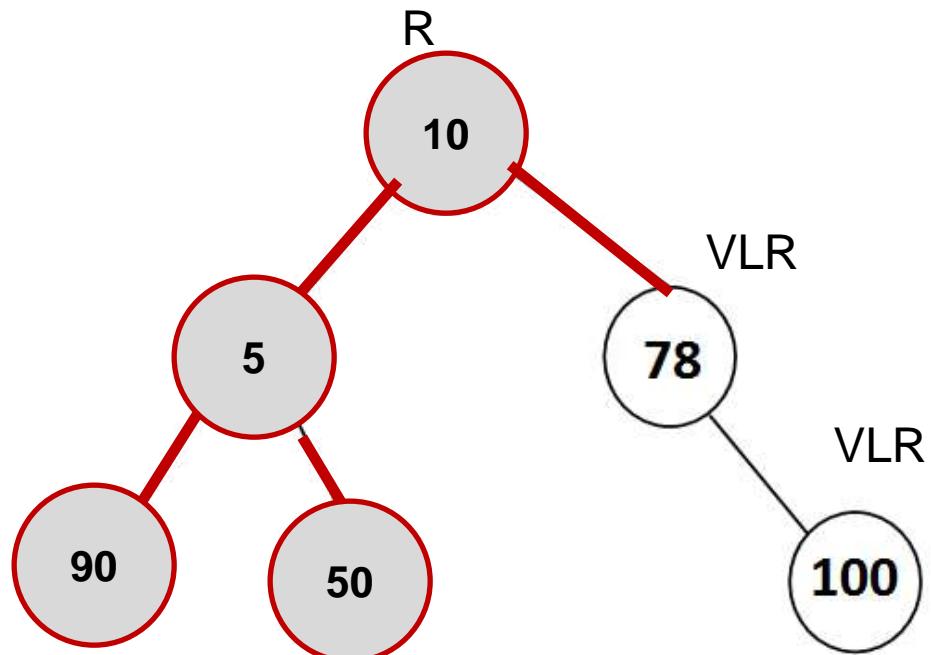
# Pre order Traversal (VLR)



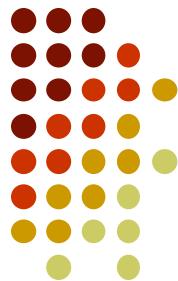
10 5 90 50



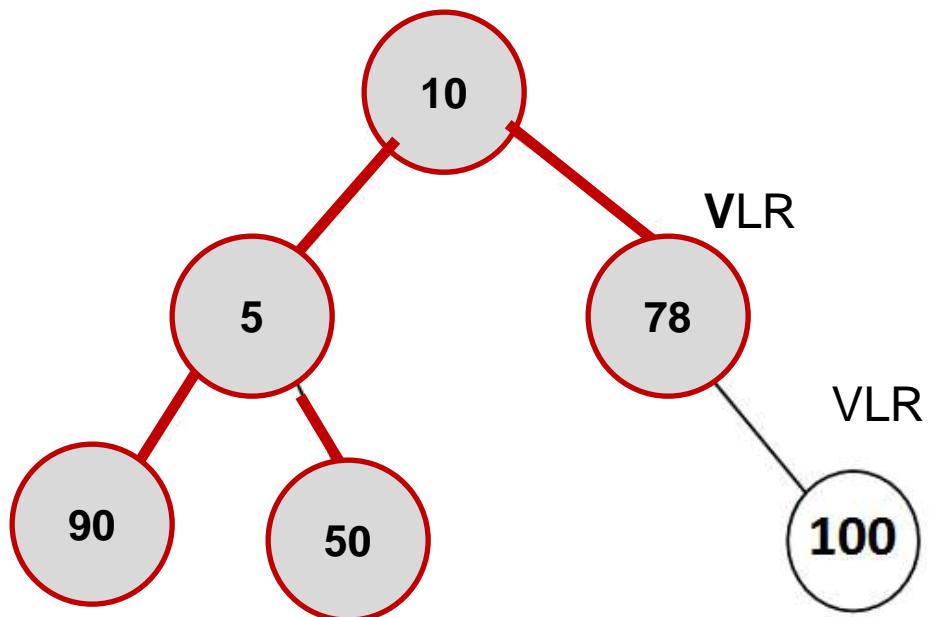
# Pre order Traversal (VLR)



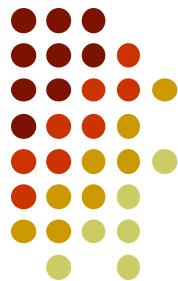
10 5 90 50



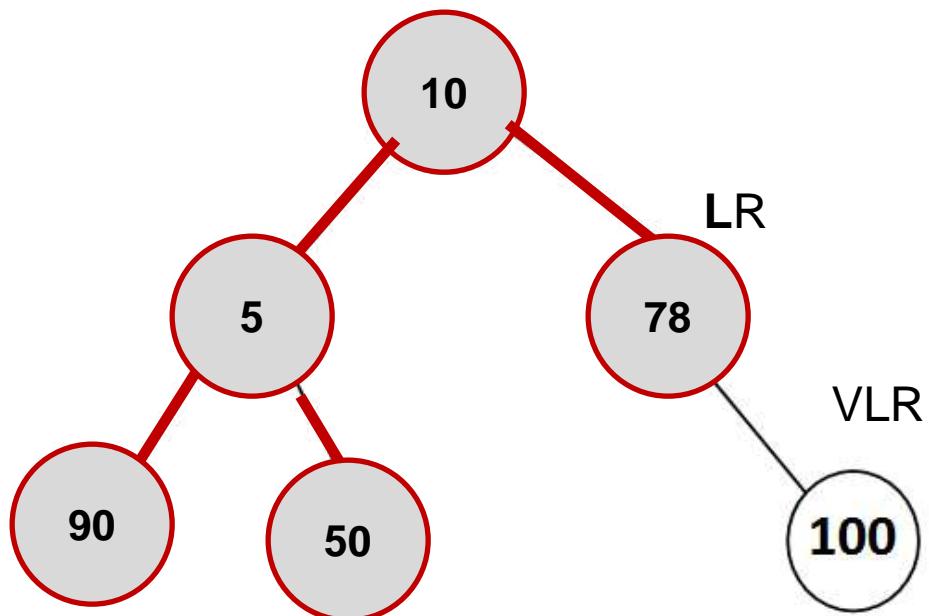
# Pre order Traversal (VLR)



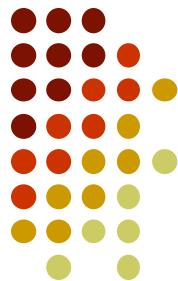
10 5 90 50 78



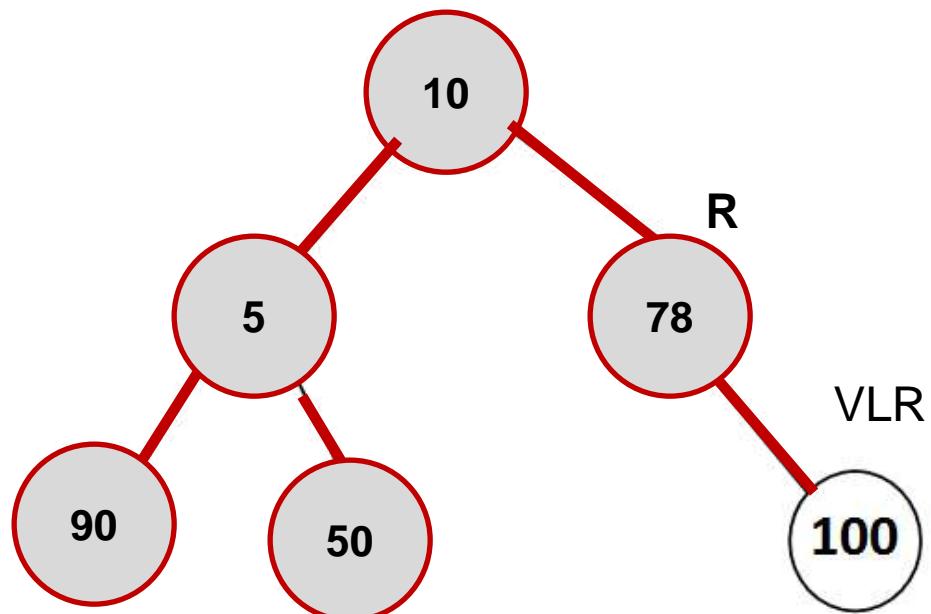
# Pre order Traversal (VLR)



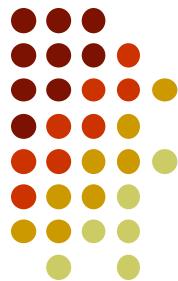
10 5 90 50 78



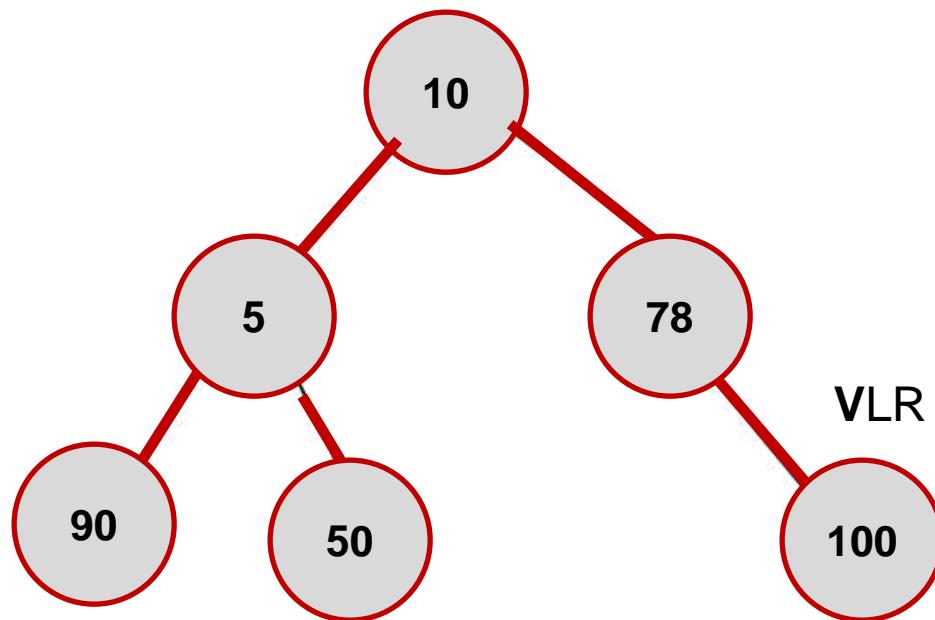
# Pre order Traversal (VLR)



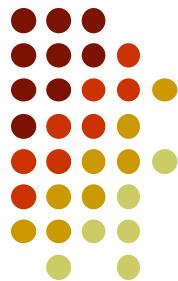
10 5 90 50 78



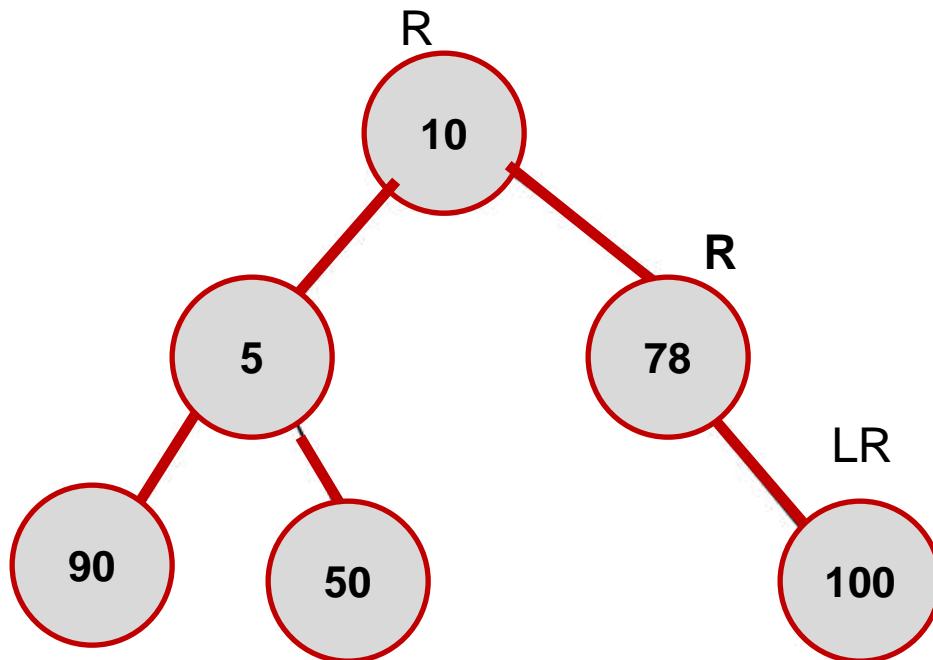
# Pre order Traversal (VLR)



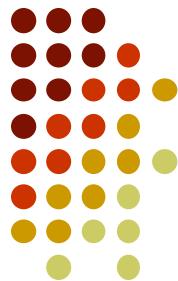
10 5 90 50 78



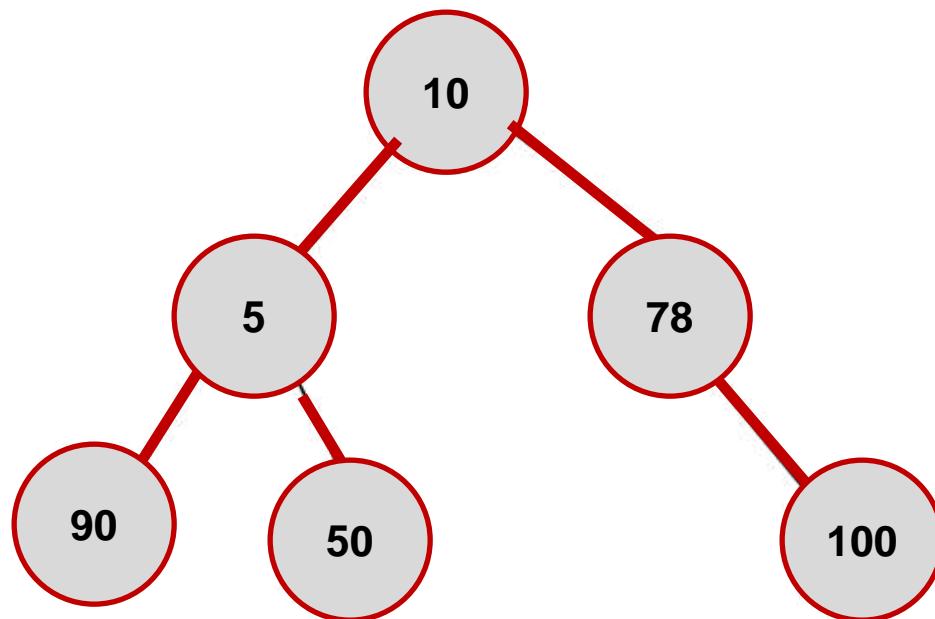
# Pre order Traversal (VLR)



10 5 90 50 78 100

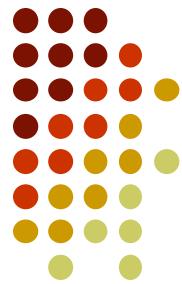


# Pre order Traversal (VLR)



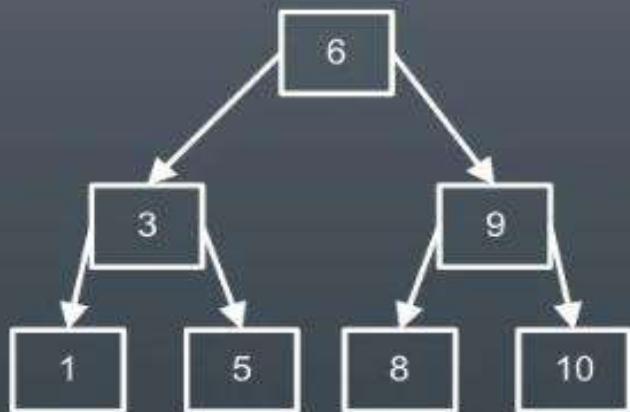
10 5 90 50 78 100

# Pre order Traversal Recursive function



```
void pre_order(root)
{
    if(root==NULL)
        return
    else
    {
        printf("%d",root->info);
        pre_order(root->lchild);
        pre_order(root->rchild);
    }
}
```

# Traversal Operation - Examples



- Inorder Traversal : 1 3 5 6 8 9 10
- Preorder Traversal : 6 3 1 5 9 8 10
- Postorder Traversal : 1 5 3 8 10 9 6

# TREES

Dr. Priyanka N

Assistant Professor Senior Grade I

School of Computer Science & Engineering  
VIT, Vellore.

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# TREES

- Introduction to Trees & Terminologies
- Binary Tree – Definition and Properties
- Binary Tree Representation
- Tree Traversals
- **Expression Trees**
- Binary Search Trees
- Operations on Binary Search Trees
  - Insertion
  - Deletion
  - Finding Max and Minimum
  - Finding the  $k^{\text{th}}$  Minimum Element

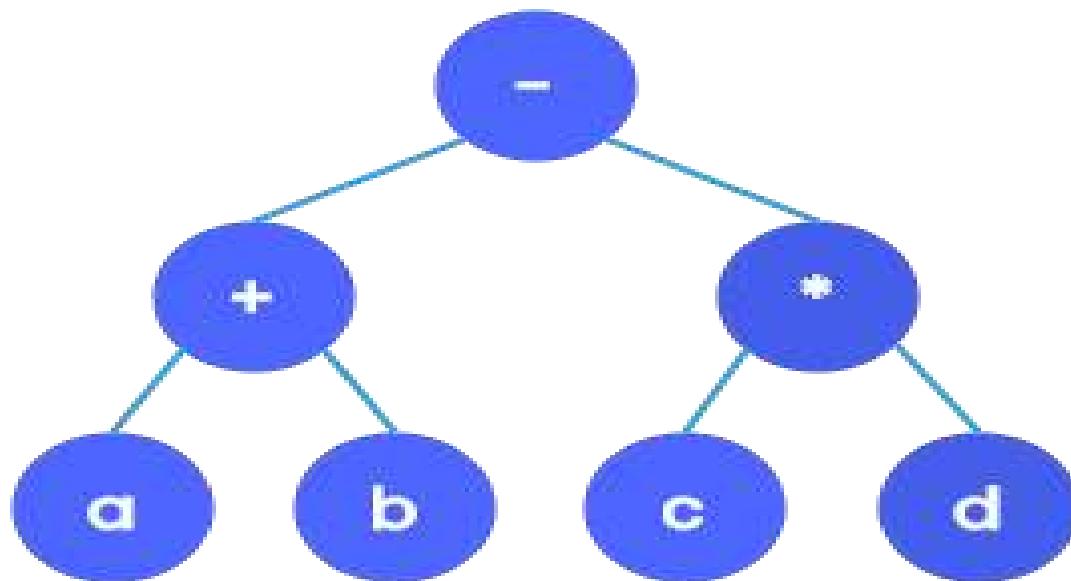
# Expression Trees

- An expression tree is a tree built up from the operands as the leaves of binary tree and operators as the non -leaves of binary tree.
- It is a special kind of binary tree in which
  - Each leaf node contains single operand
  - Each non-leaf node contains a single binary operator
  - The left and right subtrees of an operator node represent sub-expression that must be evaluated before applying the operator at the root of the subtree.

# Idea

- Levels in a binary expression tree represent the precedence of operators.
- Operators at the lower level must be evaluated first
- Then the operators at the next level and so on and at the last operator at the root node is applied and there by the expression is evaluated.

Ex :  $X = (a + b) - (c * d)$



# Constructing an Expression Tree

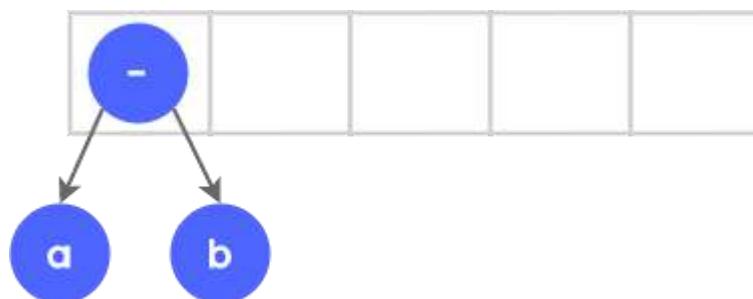
- Convert the given Expression into Postfix form.
- Repeat the following steps:
  - If the character is an operand, create a node and **PUSH** it on to the stack.
  - If the character is an operator, **POP** two values from the stack and make a node keeping the operator as root node and the two items as the left and the right child.
  - Push the newly generated node into the stack.

# Consider the Post fix Expression: a b - c \*

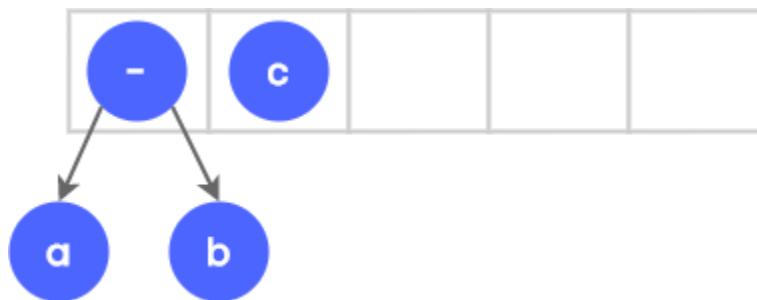
- Step I: First two characters are operands, PUSH it onto the stack.



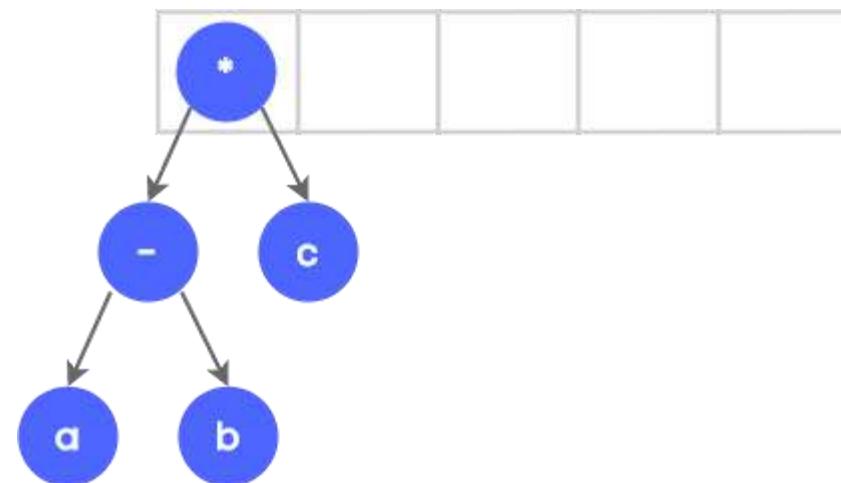
- Step 2: Next, “-” Symbol is read, “a” and “b” are popped from the stack and is added as the child of node “-”. A pointer to the new node is now pushed onto the stack.



- Step 3: Next, “c” is read and is pushed onto the stack.



- Step 4: Next, “\*”, so pop “c” and “-” from the stack and add it as child of “\*”



# Construction of Expression Tree

```
ConstructExpressionTree(postfix[])
```

```
{
```

```
// Traverse through every character of input expression
```

```
for (int i=0; i<strlen(postfix); i++)
```

```
{
```

```
// If operand, simply push into stack
```

```
if (!isOperator(postfix[i]))
```

```
{
```

```
    t = newNode(postfix[i]);
```

```
    stack.push(t);
```

```
}
```

## else // If operator

```
{  
    t = newNode(postfix[i]);  
    // Pop two top nodes  
    t1 = stack.top(); // Store top  
    stack.pop(); // Remove top  
    t2 = stack.top();  
    stack.pop();  
    // make them children to the created node  
    t->right = t1;  
    t->left = t2;  
    // Add this subexpression to stack  
    stack.push(t);  
}  
}// End of For Loop  
// only element will be root of expression tree  
t = stack.top();  
stack.pop();  
return t;  
}
```

# TREES

Dr. Priyanka N

Assistant Professor Senior Grade I

School of Computer Science & Engineering  
VIT, Vellore.

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

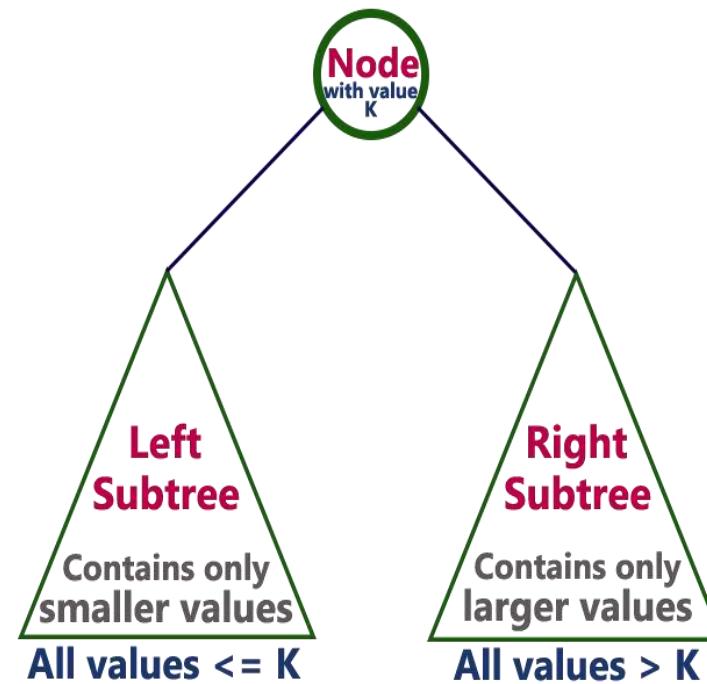
# TREES

- Introduction to Trees & Terminologies
- Binary Tree – Definition and Properties
- Binary Tree Representation
- Tree Traversals
- Expression Trees
- **Binary Search Trees**
- **Operations on Binary Search Trees**
  - Insertion
  - Deletion
  - Finding Max and Minimum
  - Finding the  $k^{\text{th}}$  Minimum Element

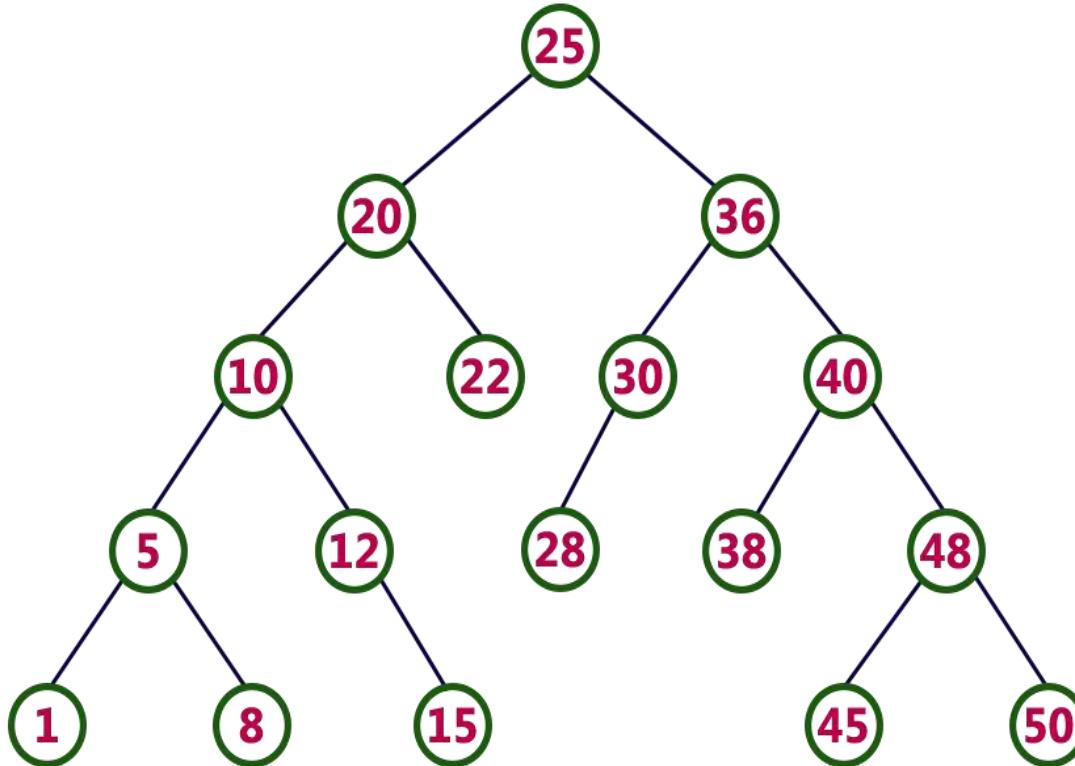
# Binary Search Trees

- Difference between Binary Tree and Binary Search Trees
  - In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values.
  - In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.
- **Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.**

# Binary Search Trees



# Example



**Every binary search tree is a binary tree but every binary tree need not to be binary search tree.**

# Operations on Binary Search Tree

- Insertion
- Deletion
- Searching
  - Finding Max and Minimum
  - Finding the  $k^{\text{th}}$  Minimum Element

# Construction of Binary Search Tree

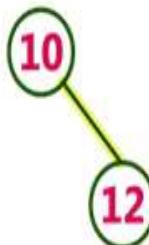
Construct a Binary Search Tree by inserting the following sequence of numbers...

**10,12,5,4,20,8,7,15 and 13**

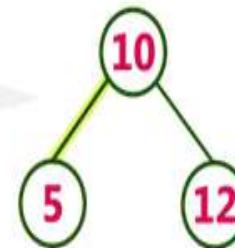
insert (10)



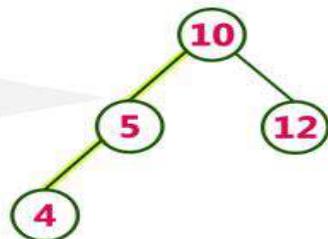
insert (12)



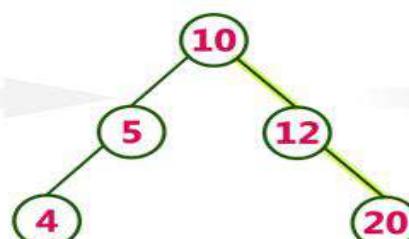
insert (5)



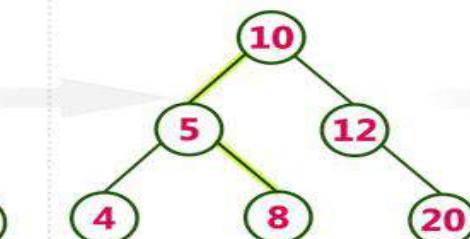
insert (4)



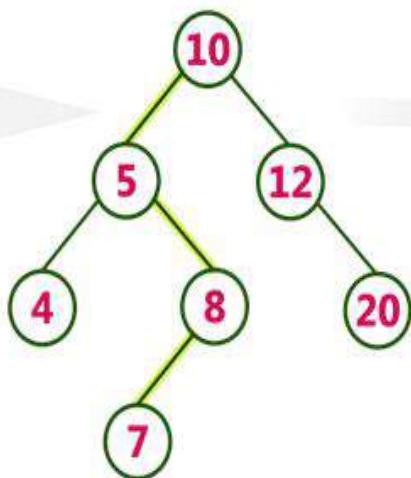
insert (20)



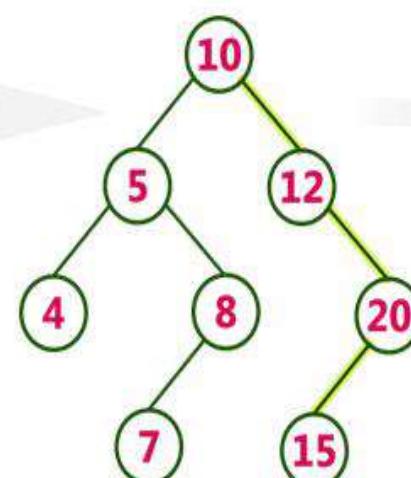
insert (8)



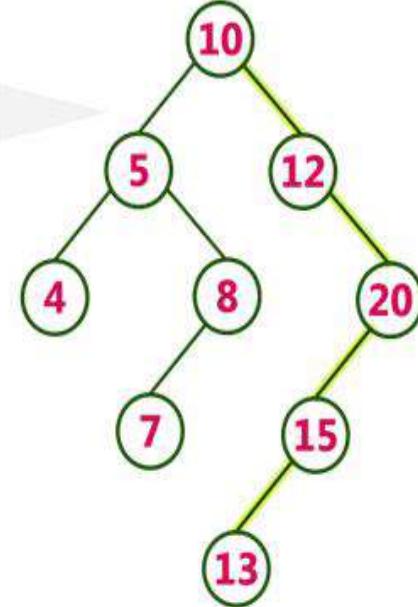
insert (7)



insert (15)



insert (13)



# Insertion Operation

- In binary search tree, new node is always inserted as a leaf node
  - Create a **newNode** with given value and set its **left** and **right** to **NULL**.
  - Check whether tree is Empty.
  - If the tree is **Empty**, then set **root to newNode**.
  - If the tree is **Not Empty**, then check whether the value of **newNode** is **smaller or larger** than the root node.
  - If **newNode** is **smaller than or equal** to the node then move to its **left child**. If **newNode** is **larger** than the node then move to its **right child**.
  - Repeat the above steps **until reaching leaf node**
  - After reaching the leaf node, insert the **newNode** as **left child** if the **newNode** is **smaller or equal** to that leaf node or else **insert it as right child**.

# Structure of Node

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

**\*root=NULL, \*tmpnode;**

# Creation of new node

```
Createnode(int value)
{
    Struct Node *newnode;
    newnode= malloc(sizeof(Struct node));
    newnode->data=value;
    newnode->left=NULL;
    newnode->right=NULL;
}
```

# Insertion Operation

```
insert(struct * root, int value)
{
    if(value == root->data)
        print "Duplicate Elements";
    else if(value < root->data)
    {
        if (root->left == NULL)
            root->left = newnode;
        else
            insert(root->left, value);
    }
    else
    {
        if(root->right == NULL)
            root->right = newnode;
        else
            insert(root->right, value);
    }
}
```

*Root and value to be inserted is passed every time*

# Search Operation

**SEARCH(KEY)**

```
{  
    struct node *temp = root;  
    while (temp != NULL)  
    {  
        if (key == temp->data)  
            return 1;  
        else if (key > temp->data)  
        {  
            temp = temp->right;  
        }  
        else  
        {  
            temp = temp->left;  
        }  
    }  
    return 0;  
}
```

# Finding Minimum

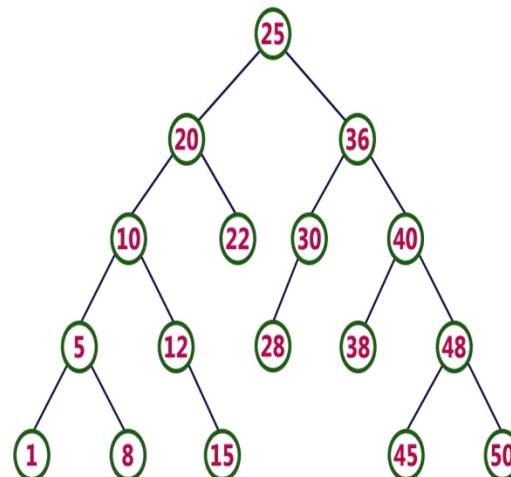
```
struct node *smallest_node(struct node *root)
{
    struct node *q = root;
    while (q != NULL && q->left != NULL)
    {
        q = q->left;
    }
    return q;
}
```

# Finding Maximum

```
struct node *largest_node(struct node *root)
{
    struct node *q = root;
    while (q!= NULL && q->right != NULL)
    {
        q = q->right;
    }
    return q;
}
```

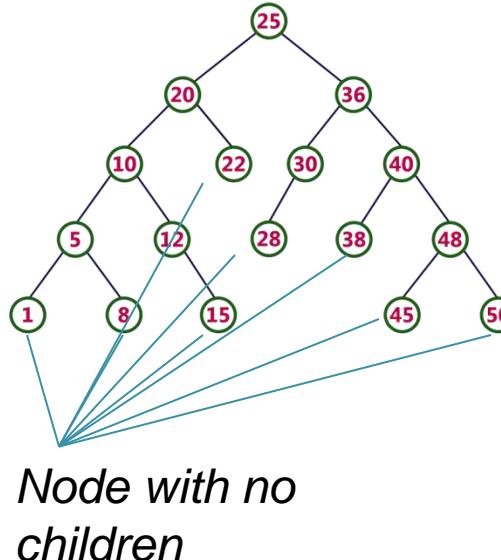
# Deletion Operation

- In binary search tree, Deletion is carried out in Three cases
  - Case I: Deleting a Leaf node (A node with no children)
  - Case 2: Deleting a node with one child
  - Case 3: Deleting a node with two children



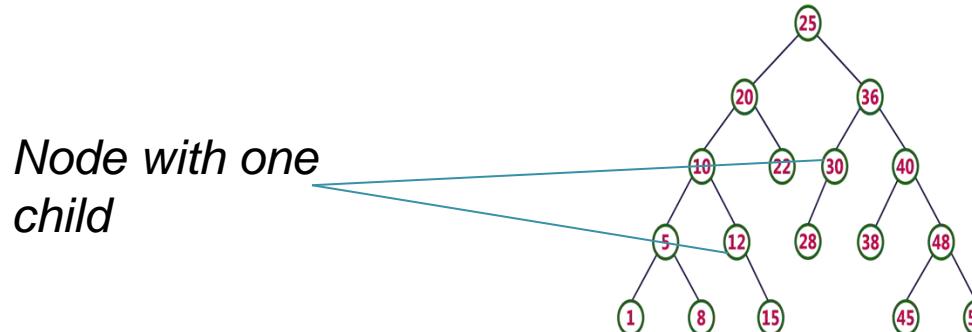
# Deletion Operation – CASE I

- Case I: Deleting a Leaf node (A node with no children)
  - Step 1 - Find the node to be deleted using search operation
  - Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.



# Deletion Operation – CASE II

- **Case 2: Deleting a node with one child**
  - Step 1: Find the node to be deleted using search operation
  - Step 2 : If it has only one child then create a link between its parent node and child node.
  - Step 3: Delete the node using free function and terminate the function.



# Deletion Operation – CASE III

- **Case 3: Deleting a node with two children**

- Step 1 - Find the node to be found using search operation
- Step 2 - If it has two children, then find the largest node in its left subtree (**OR**) the smallest node in its right subtree.
- Step 3 - Swap both **deleting node** and **node** which is found in the above step.
- Step 4 - Then check whether deleting node came to case 1 or case 2 or else go to step 2
- Step 5 - If it comes to case 1, then delete using case 1 logic.
- Step 6- If it comes to case 2, then delete using case 2 logic.
- Step 7 - Repeat the same process until the node is deleted from the tree.

```
deleteNode(struct node* root, int key)
{
    if (root == NULL)
        return root;

    // key less than value of root then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // key greater than value of root then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // key is same as value of root, node be deleted is found
    else
    {
```

**// node with only one child or no child**

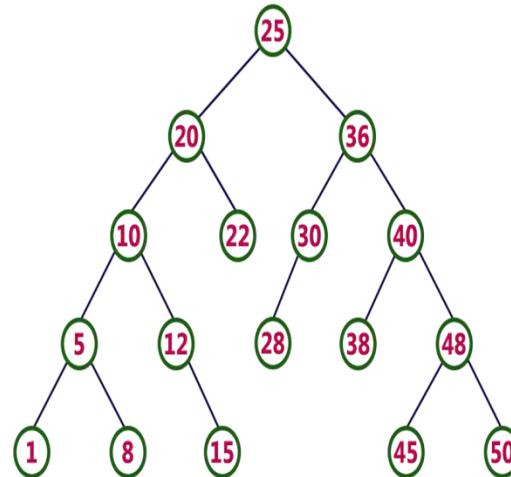
```
if (root->left == NULL)
{
    struct node* temp = root->right;
    free(root);
    return temp;
}

else if (root->right == NULL)
{
    struct node* temp = root->left;
    free(root);
    return temp;
}
```

```
// node with two children, Get smallest in  
// the right subtree)  
struct node* temp = minValueNode(root->right);  
root->key = temp->key;  
// Delete the node  
root->right = deleteNode(root->right, temp->key);  
}  
return root;  
}
```

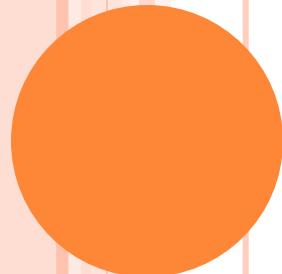
# Practice Yourself

- Finding the  $k^{\text{th}}$  Minimum Element

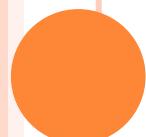


Example: 5<sup>th</sup> Minimum Element in the Tree?? 12

8<sup>th</sup> Minimum Element in the Tree?? 22



HEAPS



Dr.Priyanka N



# GOALS

- To explore the implementation, testing and performance of heap sort algorithm



# HEAP

- A heap is a data structure that stores a collection of objects (with keys), and has the following properties:
  - Complete Binary tree
  - Heap Order
- It is implemented as an array where each node in the tree corresponds to an element of the array.
- An array is viewed as a nearly complete binary tree
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)



# HEAPS



A **heap** is a certain kind of complete binary tree.

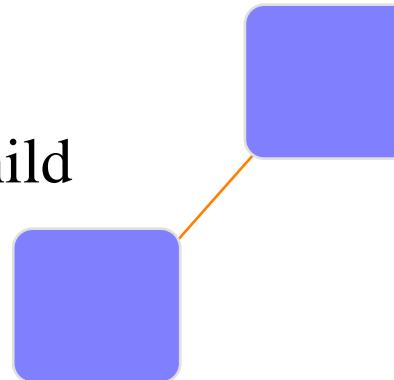
When a complete binary tree is built, its first node must be the root.



# HEAPS

Complete binary tree.

Left child  
of the  
root

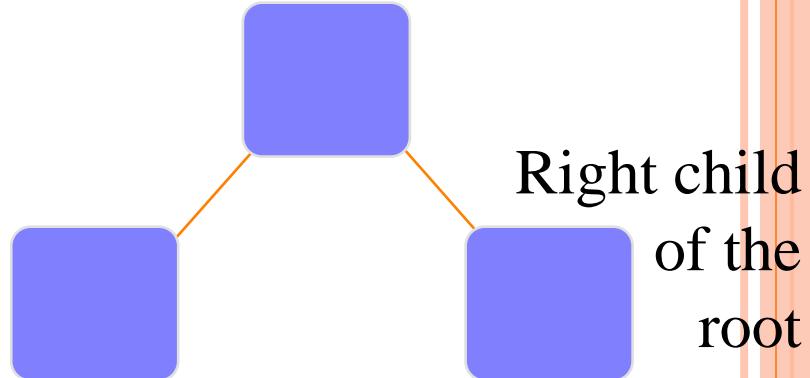


The second node is  
always the left child  
of the root.



# HEAPS

Complete binary tree.

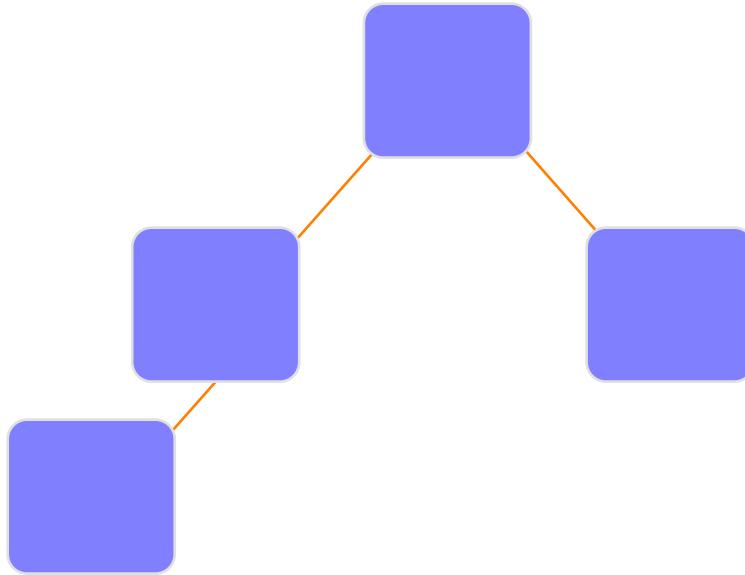


The third node is  
always the right child  
of the root.



# HEAPS

Complete binary tree.

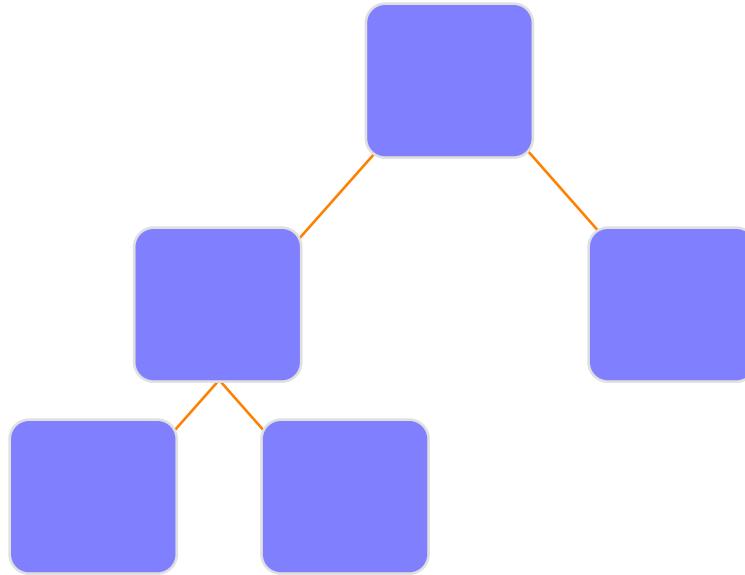


The next nodes  
always fill the next  
level from left-to-right.



# HEAPS

Complete binary tree.

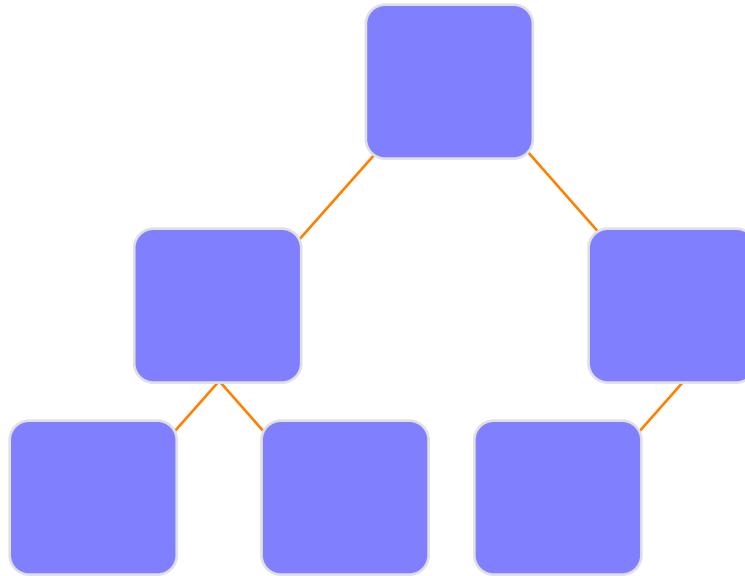


The next nodes  
always fill the next  
level from left-to-right.



# HEAPS

Complete binary tree.

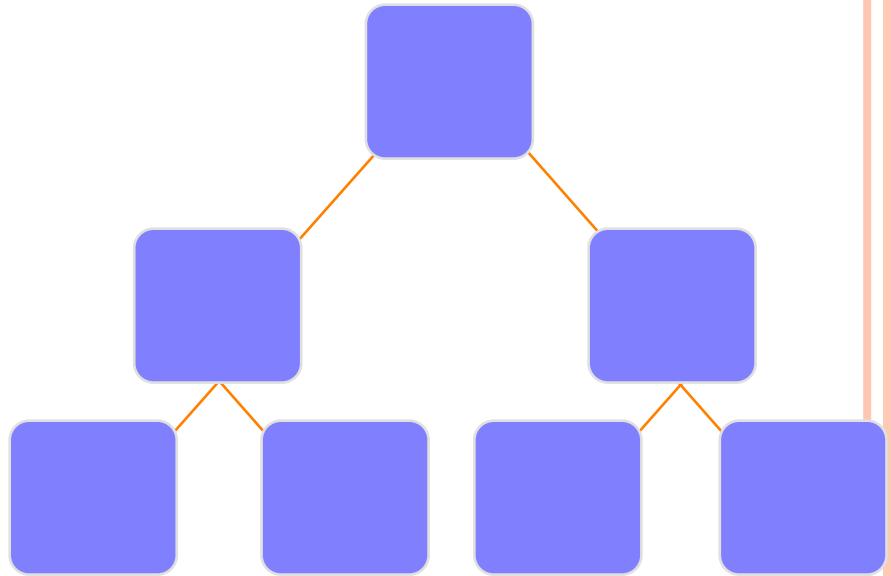


The next nodes  
always fill the next  
level from left-to-right.



# HEAPS

Complete binary tree.

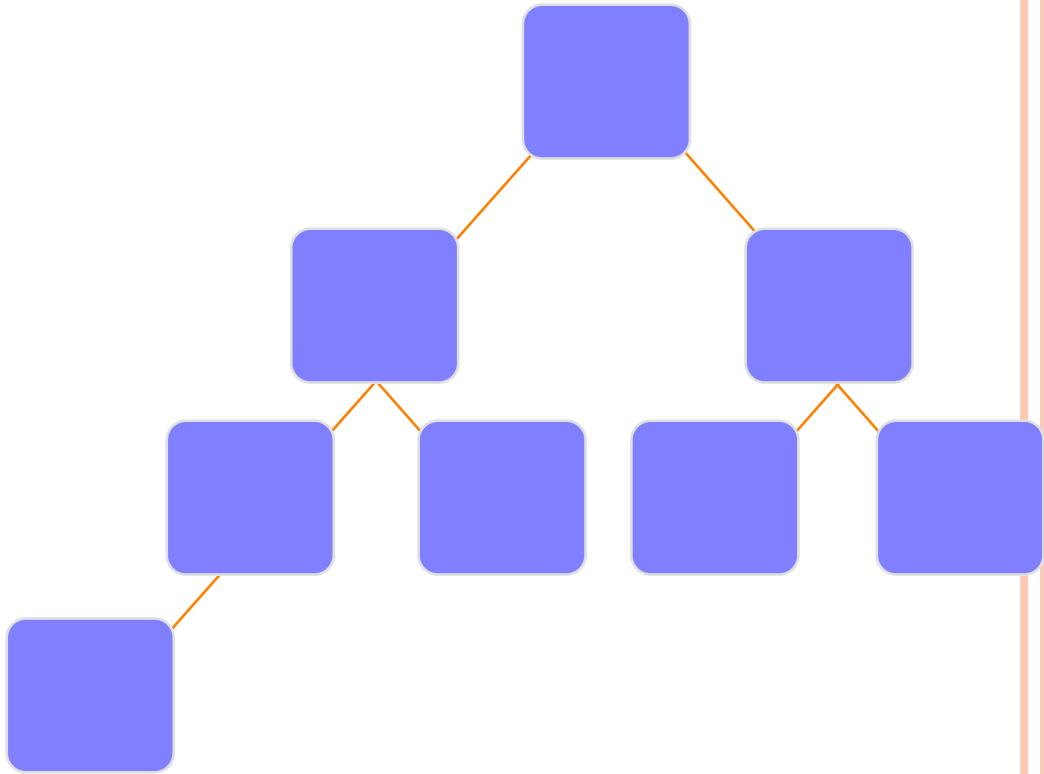


The next nodes  
always fill the next  
level from left-to-right.



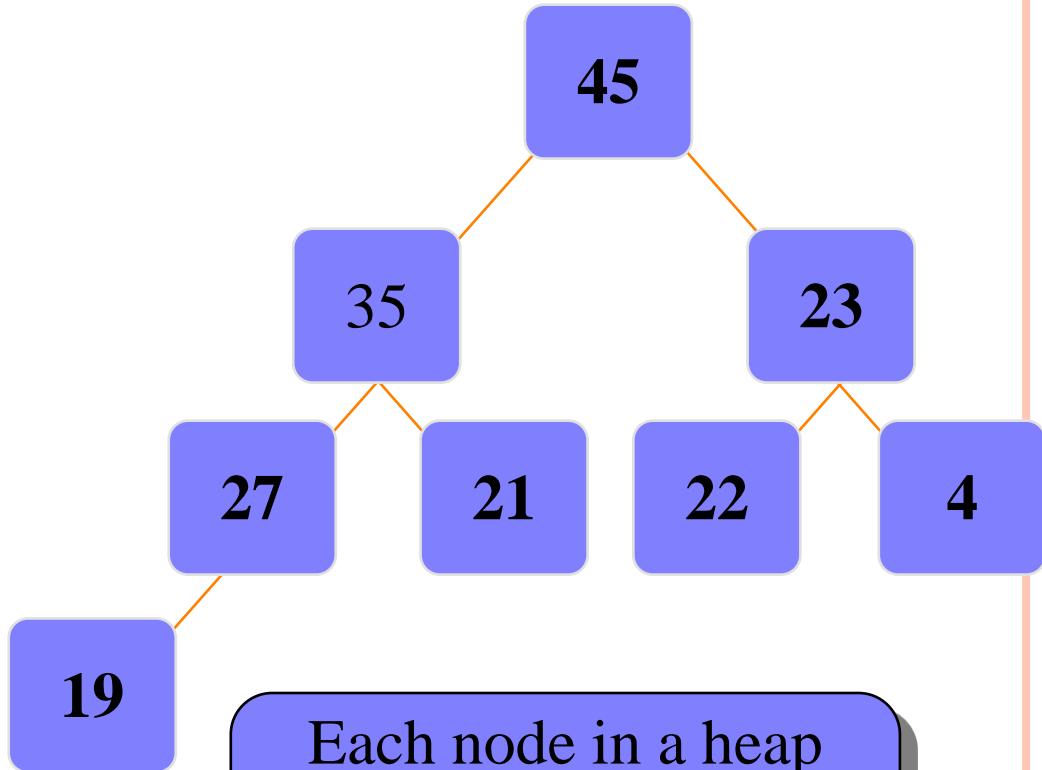
# HEAPS

Complete binary tree.



# HEAPS

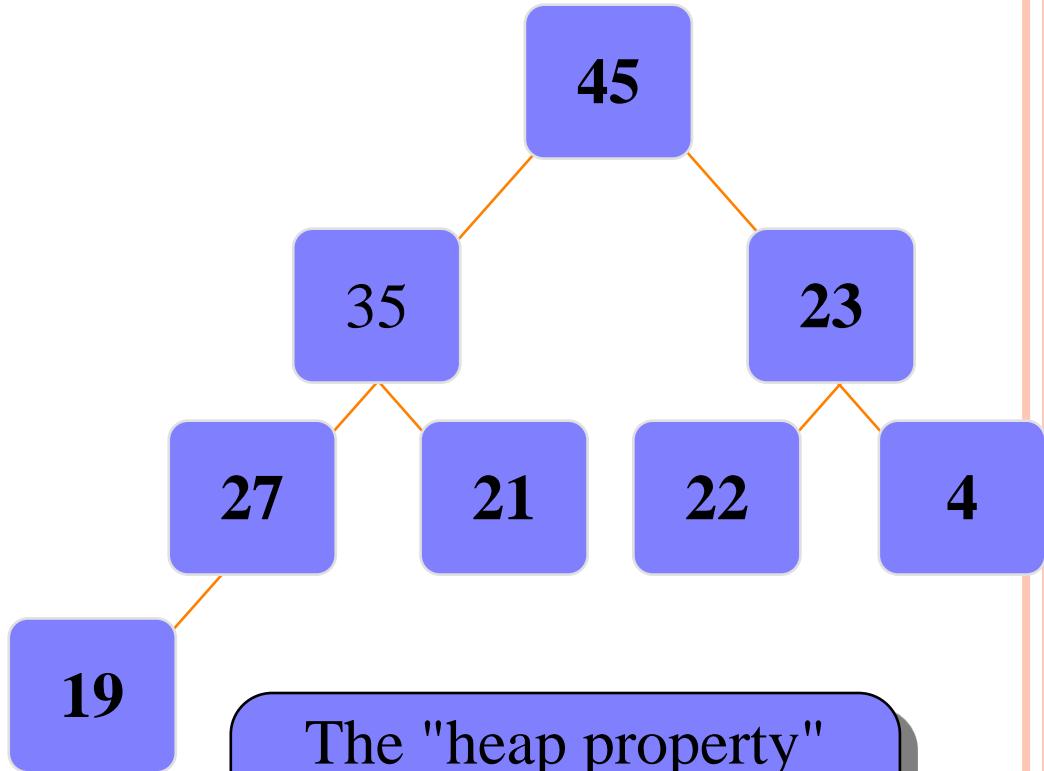
A heap is a certain kind of complete binary tree.



Each node in a heap contains a key that can be compared to other nodes' keys.

# HEAPS

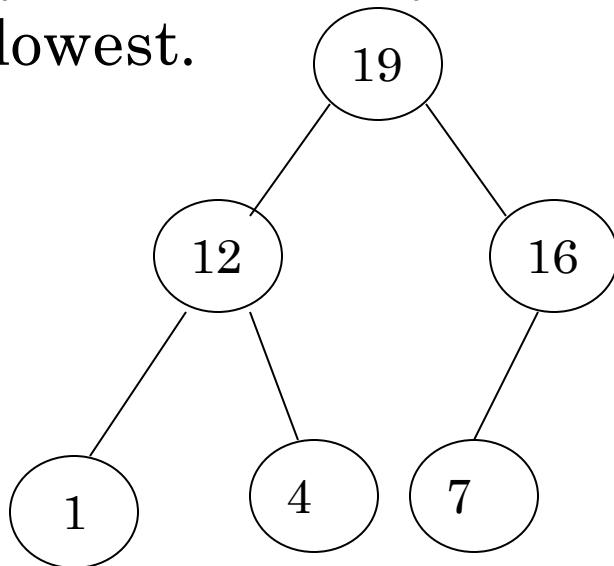
A heap is a certain kind of complete binary tree.



The "heap property" requires that each node's key is  $\geq$  the keys of its children

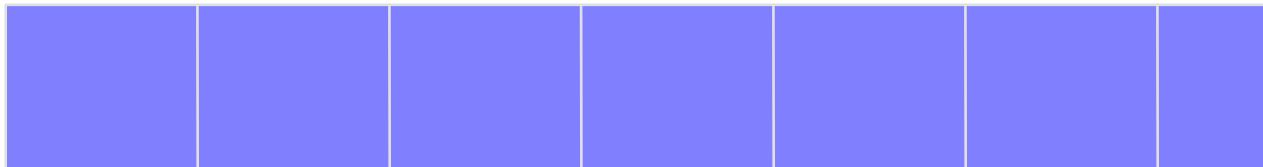
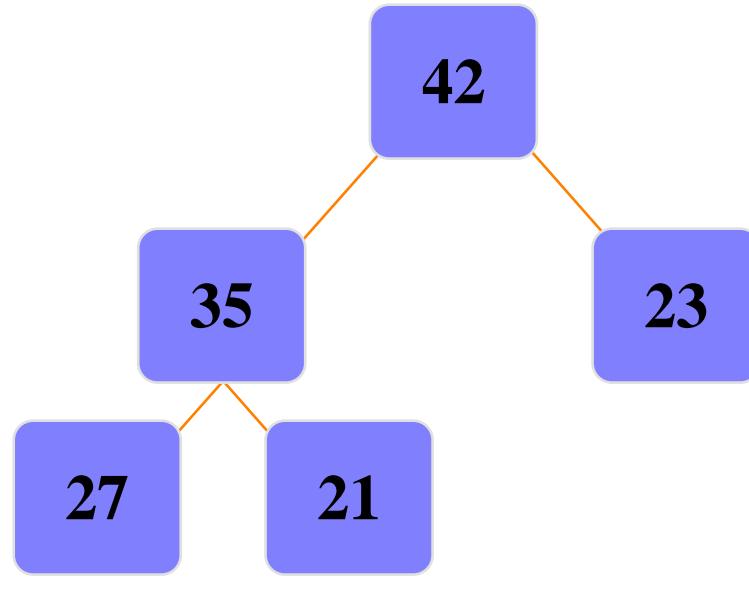
# HEAP- DEFINITION

- The binary heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array.
- The array is completely filled on all levels except possibly lowest.



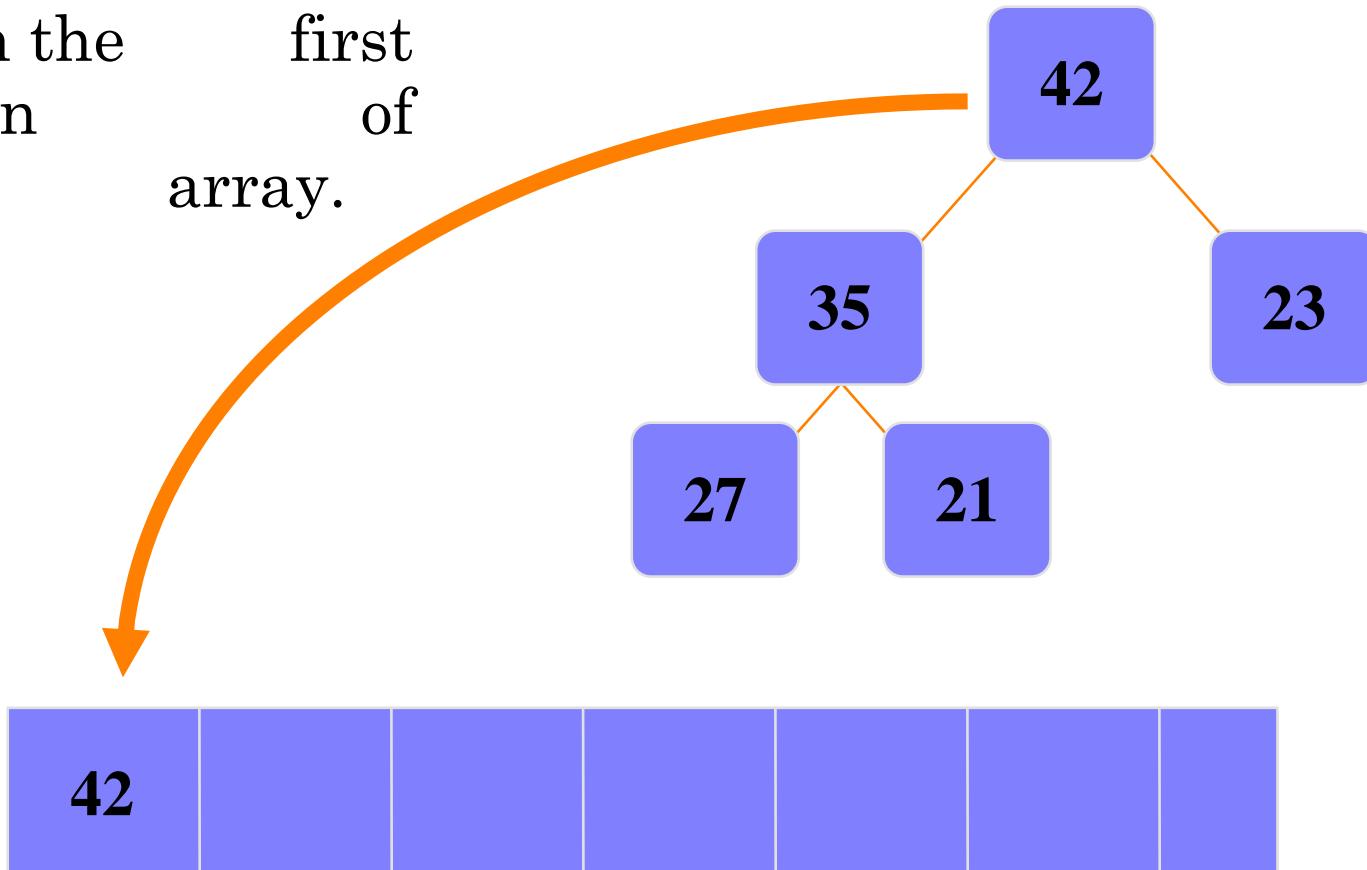
# IMPLEMENTING A HEAP

- We will store the data from the nodes in a partially-filled array.



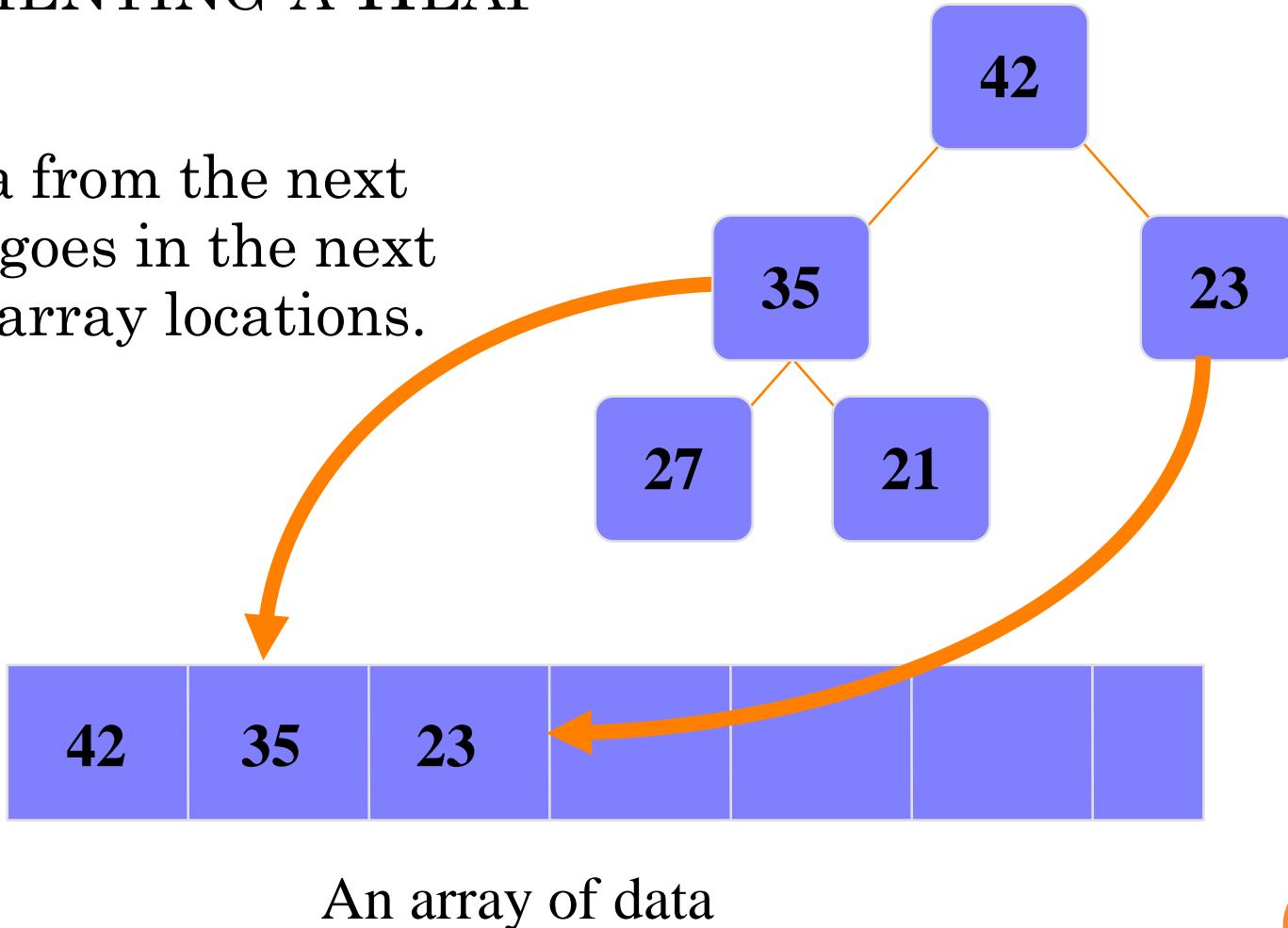
# IMPLEMENTING A HEAP

- Data from the root goes in the first location of the array.



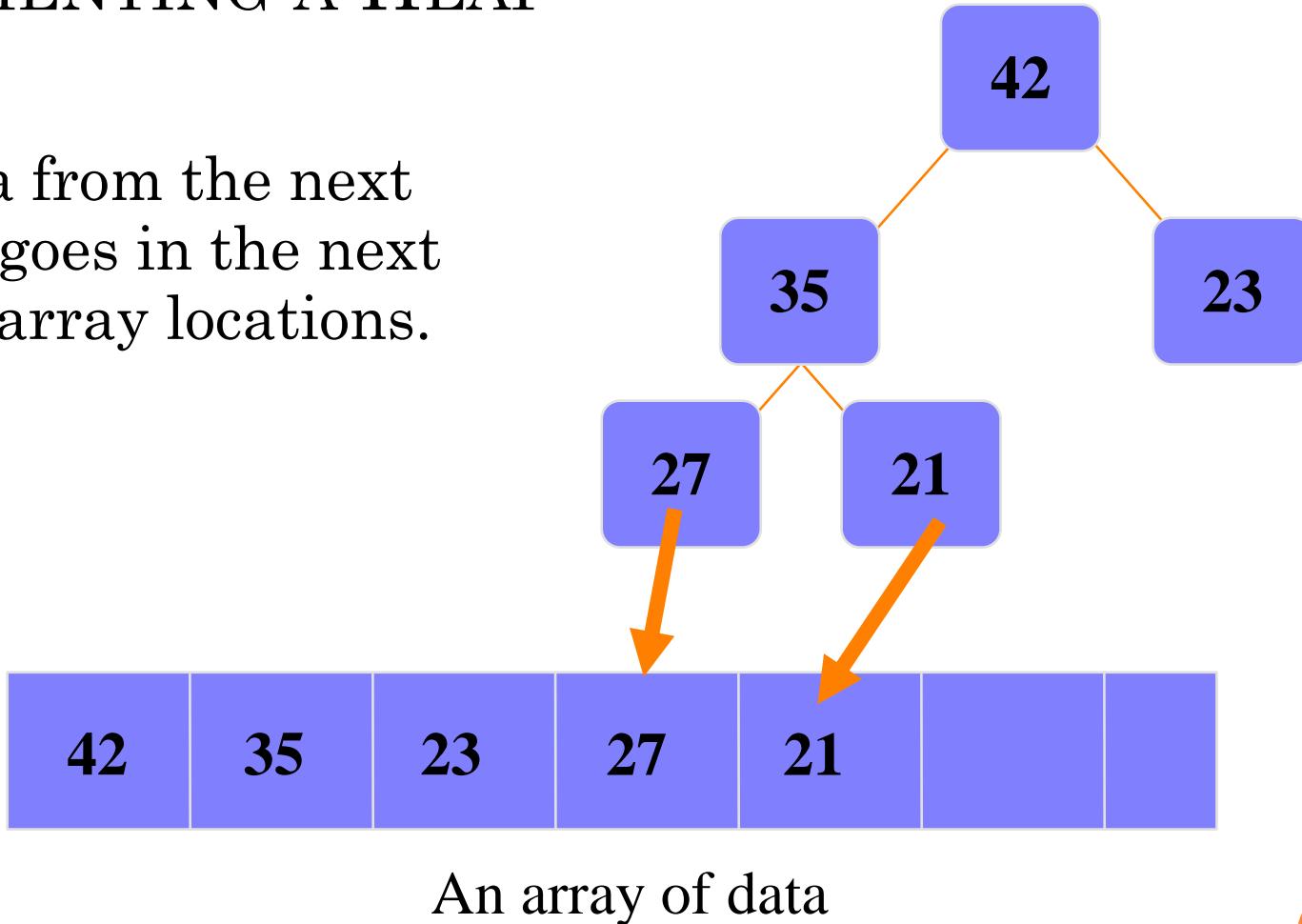
# IMPLEMENTING A HEAP

- Data from the next row goes in the next two array locations.



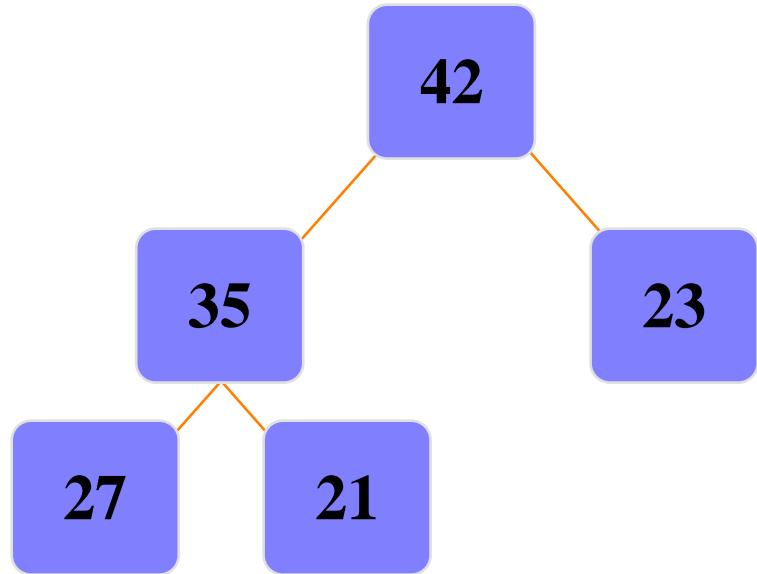
# IMPLEMENTING A HEAP

- Data from the next row goes in the next two array locations.



# IMPLEMENTING A HEAP

- Data from the next row goes in the next two array locations.

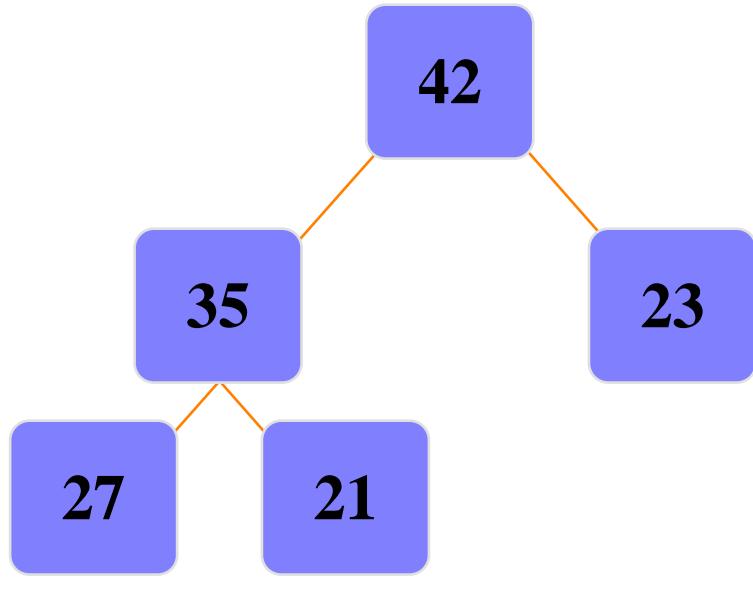


An array of data

We don't care what's in  
this part of the array.

# IMPORTANT POINTS ABOUT THE IMPLEMENTATION

- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

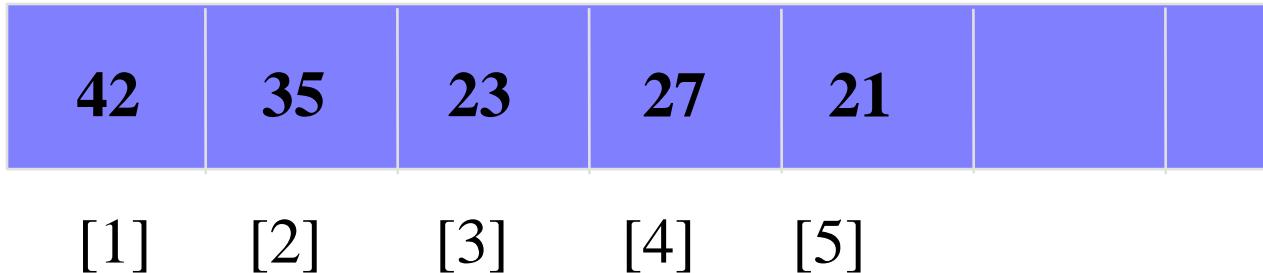
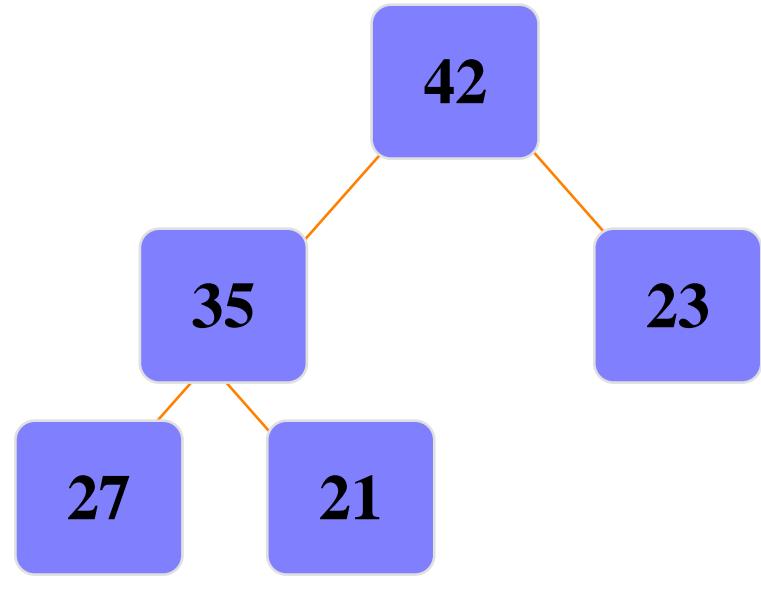


An array of data



# IMPORTANT POINTS ABOUT THE IMPLEMENTATION

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.



# HEAP

- The root of the tree A[1] and given index  $i$  of a node, the indices of its parent, left child and right child can be computed

PARENT ( $i$ )

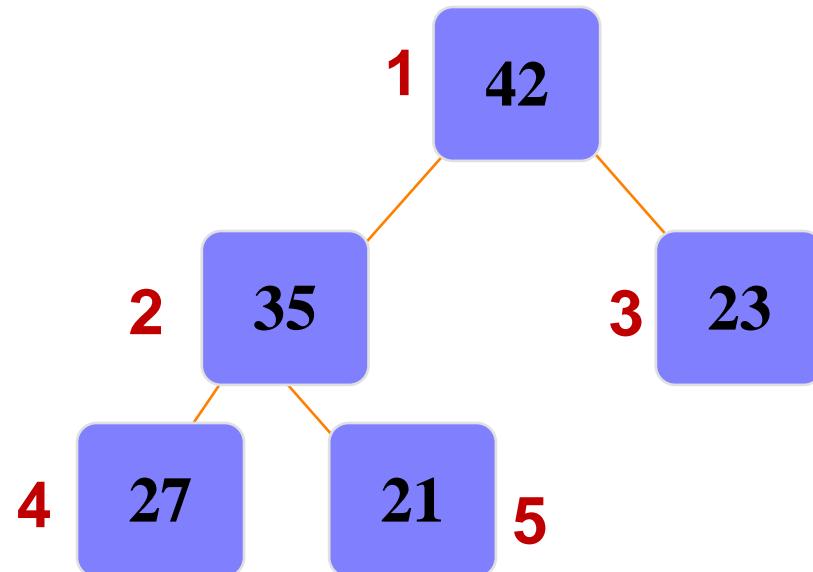
return  $\text{floor}(i/2)$

LEFT ( $i$ )

return  $2i$

RIGHT ( $i$ )

return  $2i + 1$



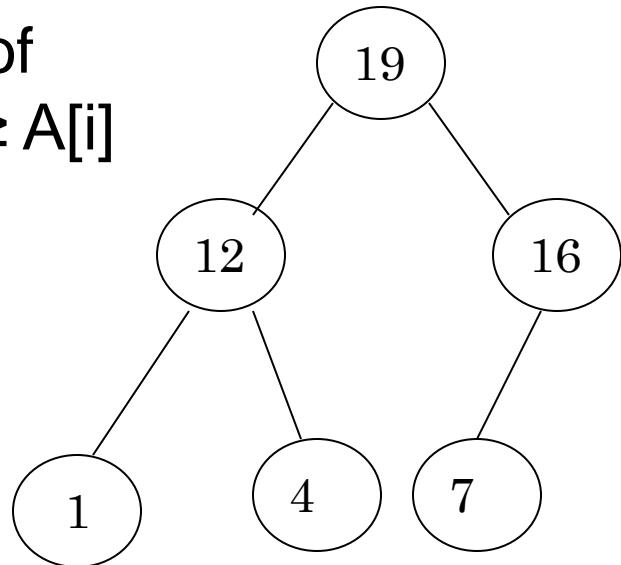
## HEAP ORDER PROPERTY

- For every node  $v$ , other than the root, the key stored in  $v$  is greater or equal (smaller or equal for max heap) than the key stored in the parent of  $v$ .
- In this case the maximum value is stored in the root



## MAX HEAP EXAMPLE

- Store data in ascending order
- Has property of  $A[\text{Parent}(i)] \geq A[i]$



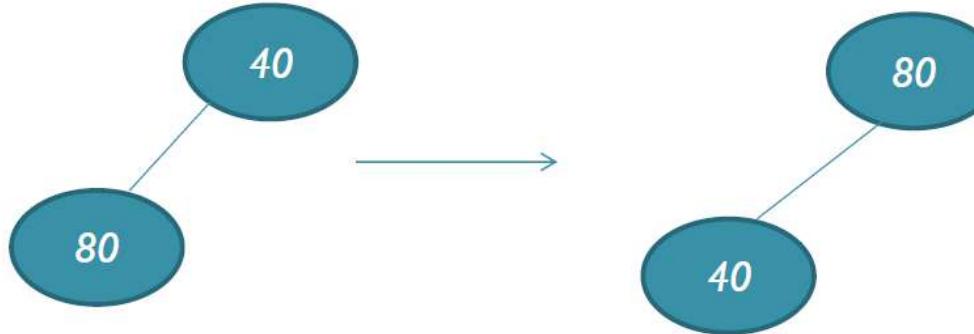
# CONSTRUCTION OF MAX HEAP - EXAMPLE

- Input data = {40,80,35,90, 45,50,70}

- Insert 40

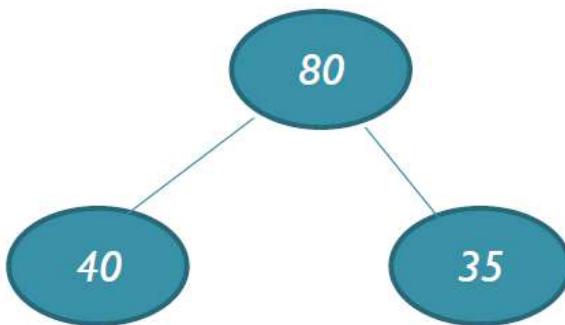


- Insert 80

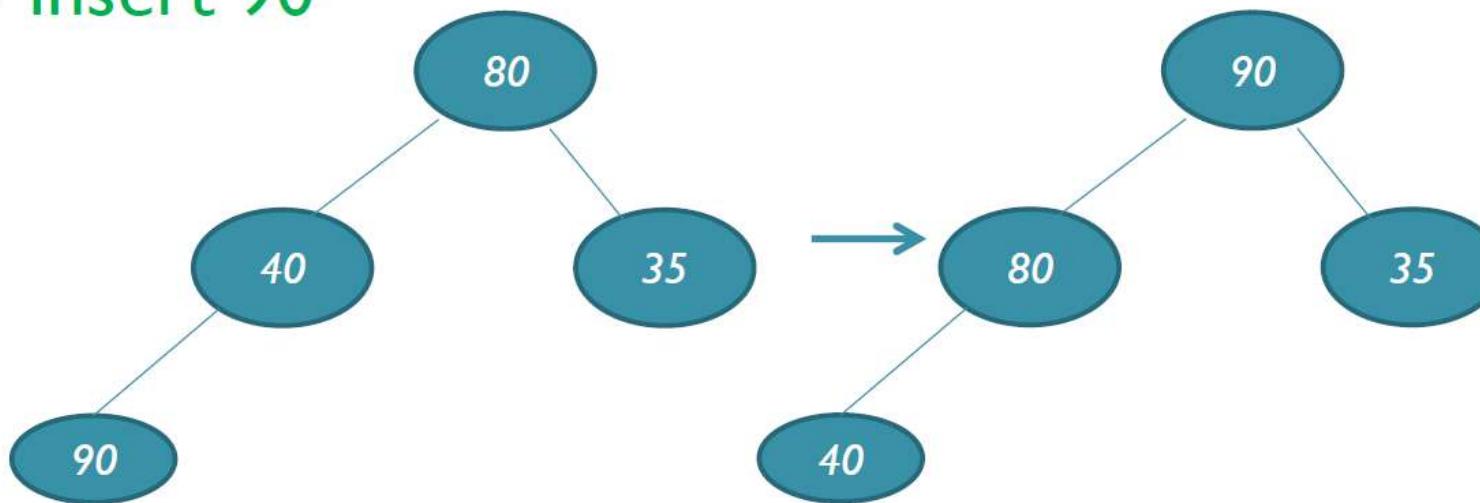


# CONSTRUCTION OF MAX HEAP - EXAMPLE

- Insert 35

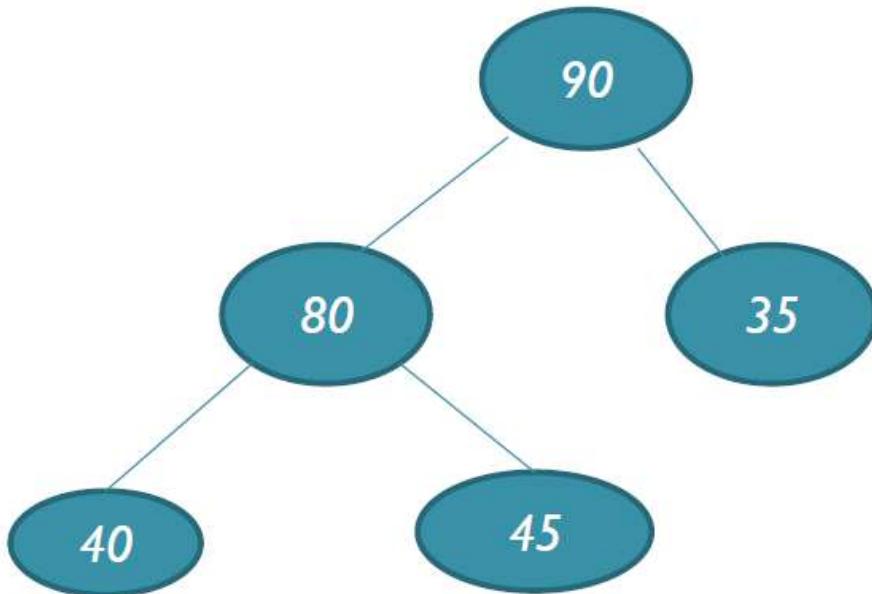


- Insert 90



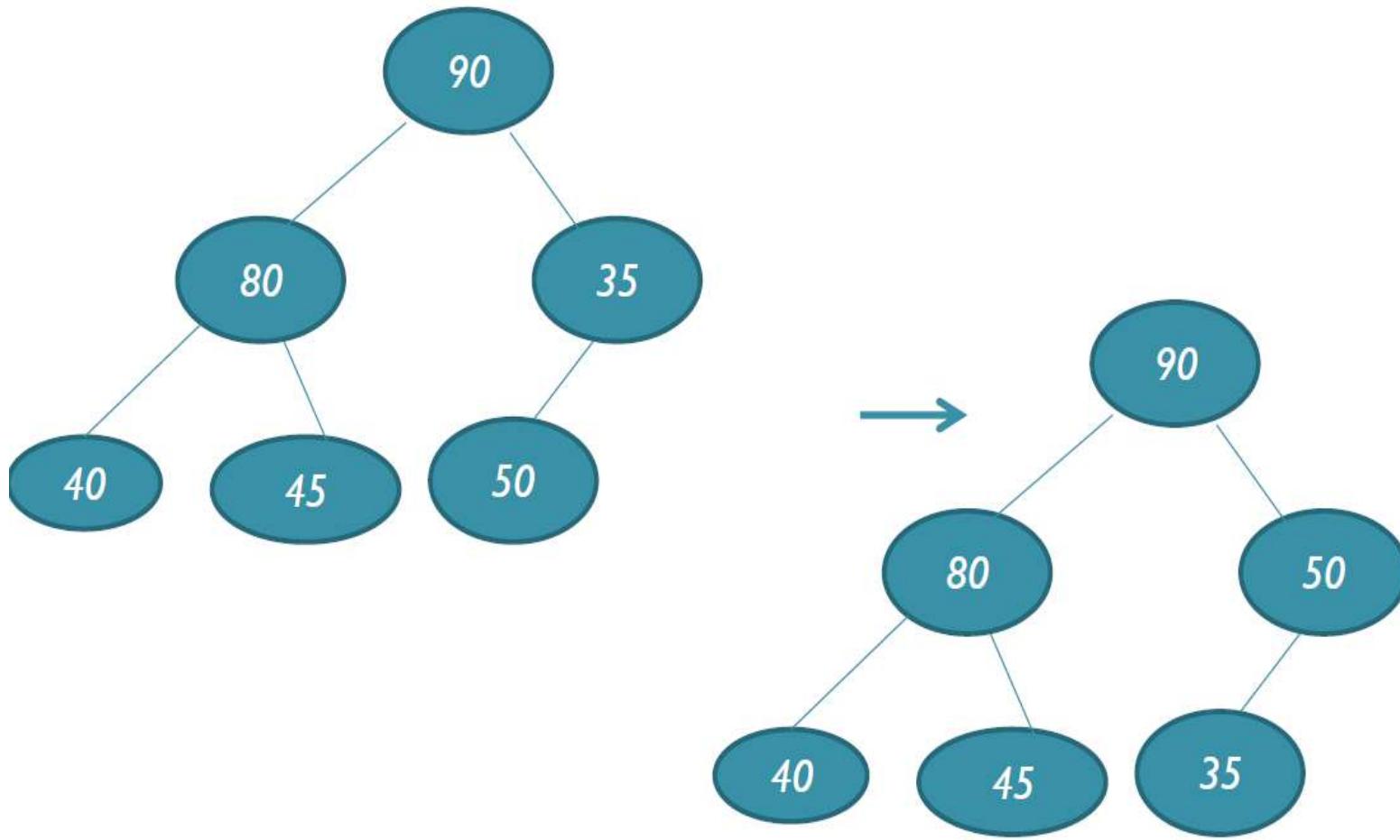
# CONSTRUCTION OF MAX HEAP - EXAMPLE

- Insert 45

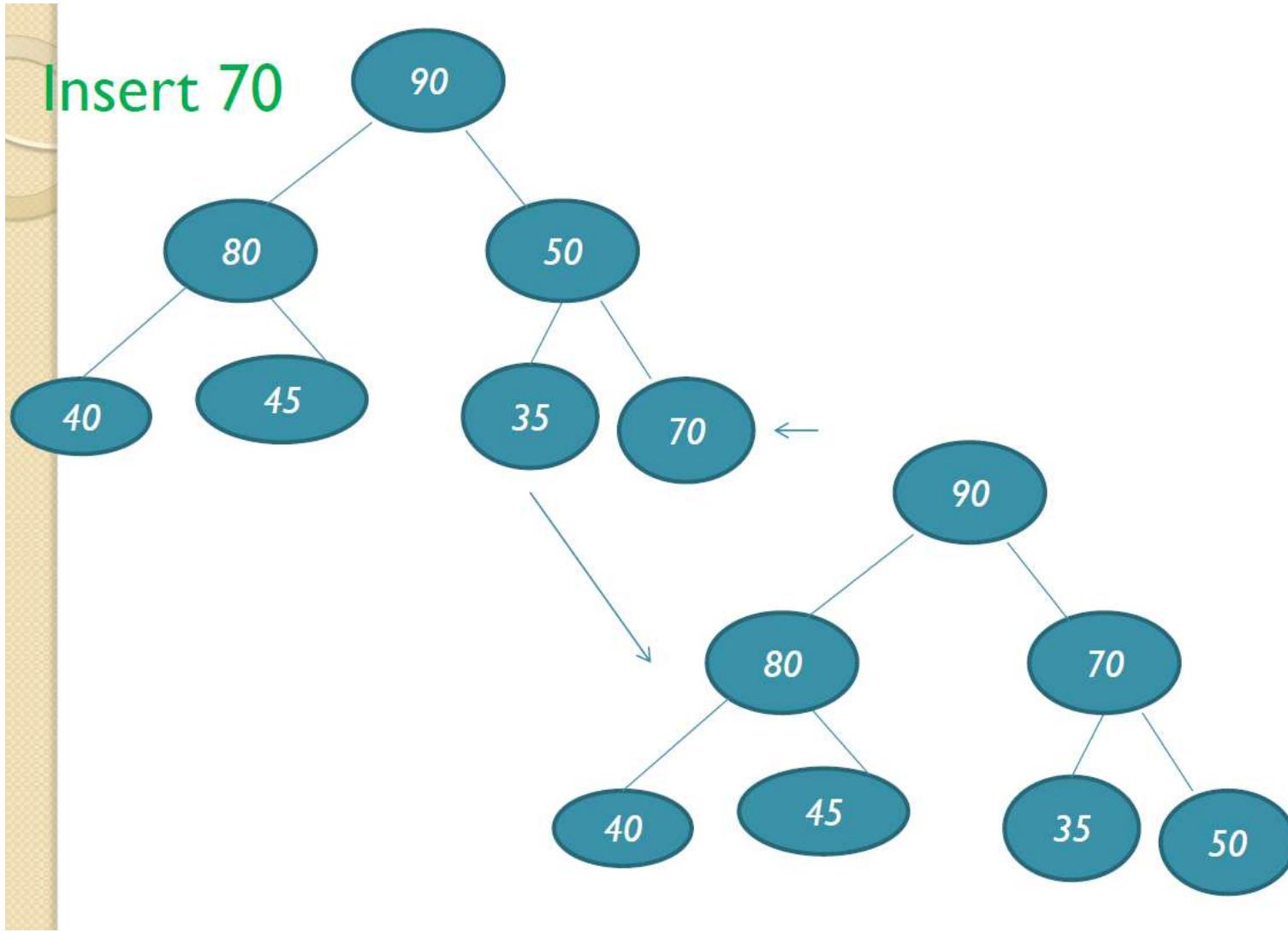


# CONSTRUCTION OF MAX HEAP - EXAMPLE

- Insert 50



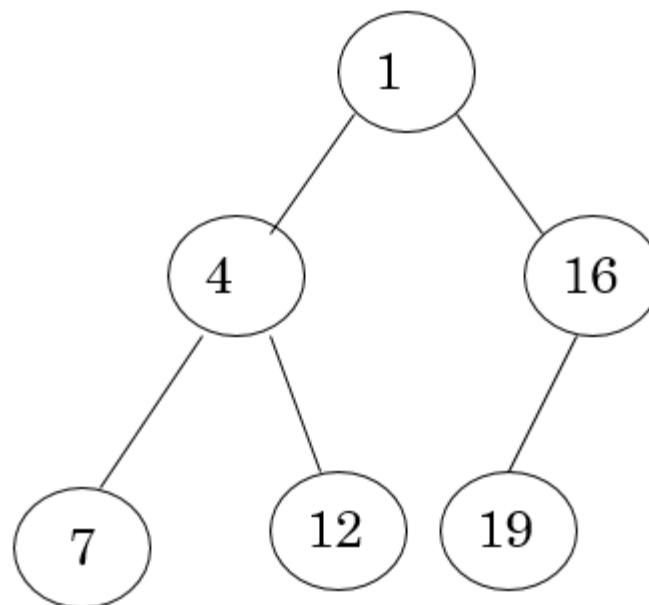
# CONSTRUCTION OF MAX HEAP - EXAMPLE



# DEFINITION

- Min Heap

- Store data in descending order
- Has property of  
 $A[\text{Parent}(i)] \leq A[i]$



# INSERTION

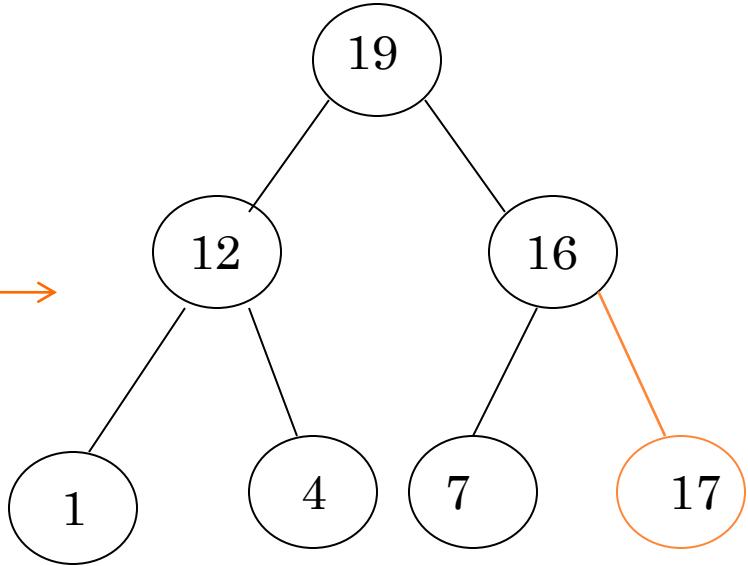
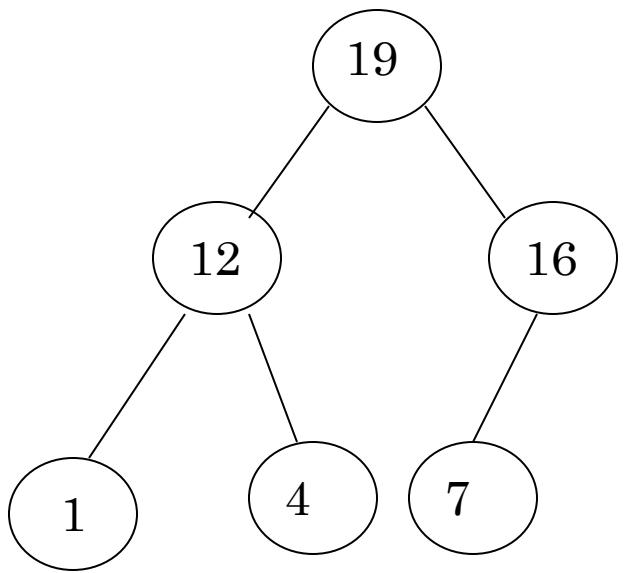
- Algorithm
  - 1. Add the new element to the next available position at the lowest level
  - 2. Restore the max-heap property if violated
    - General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child.

OR

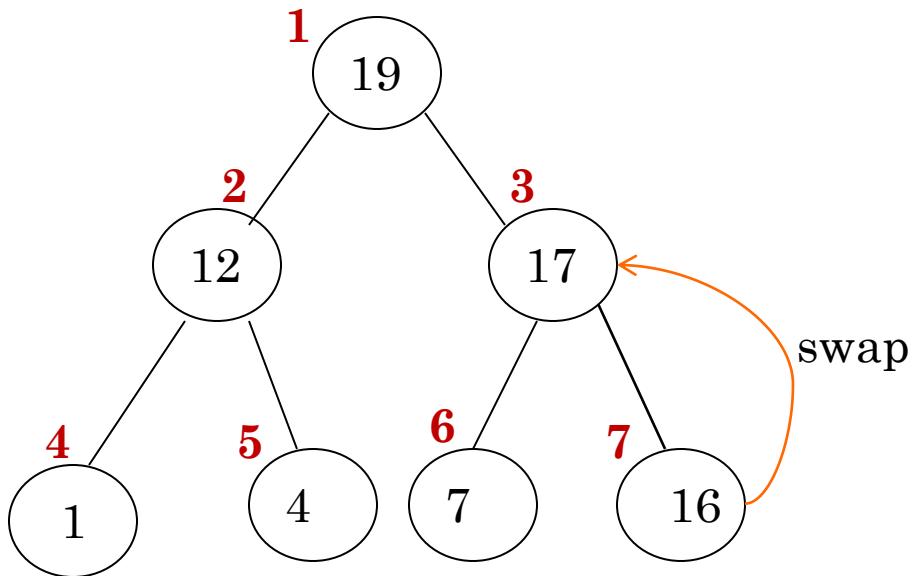
Restore the min-heap property if violated

- General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.

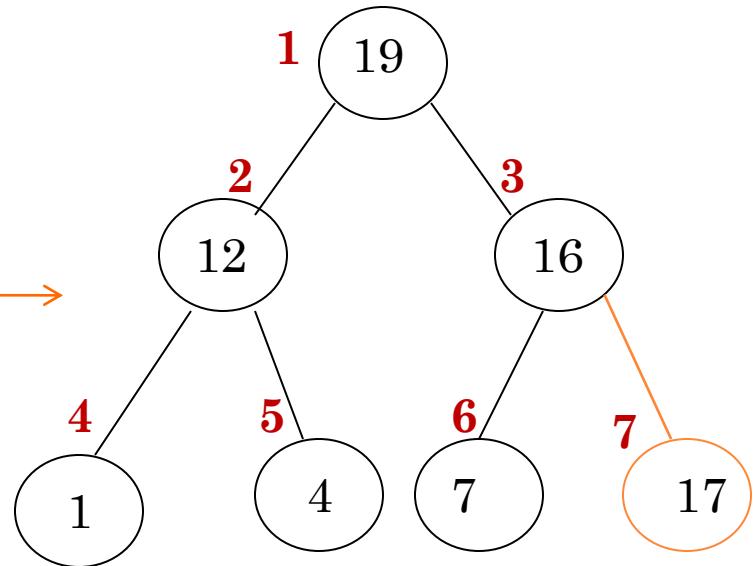
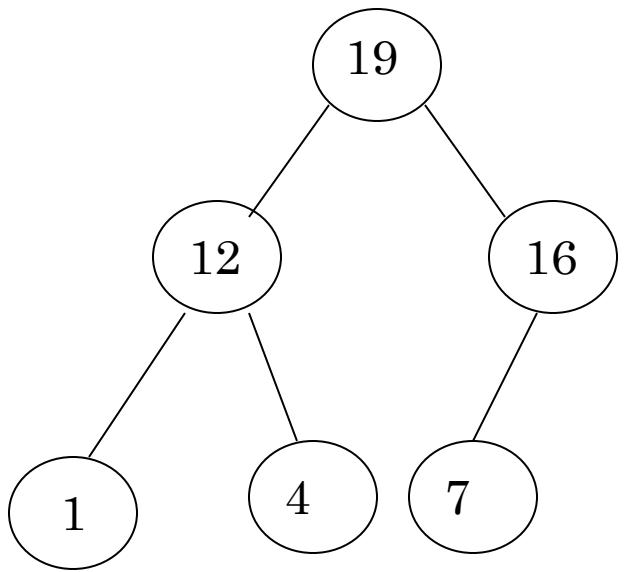




Insert 17



Percolate up to maintain the  
heap property



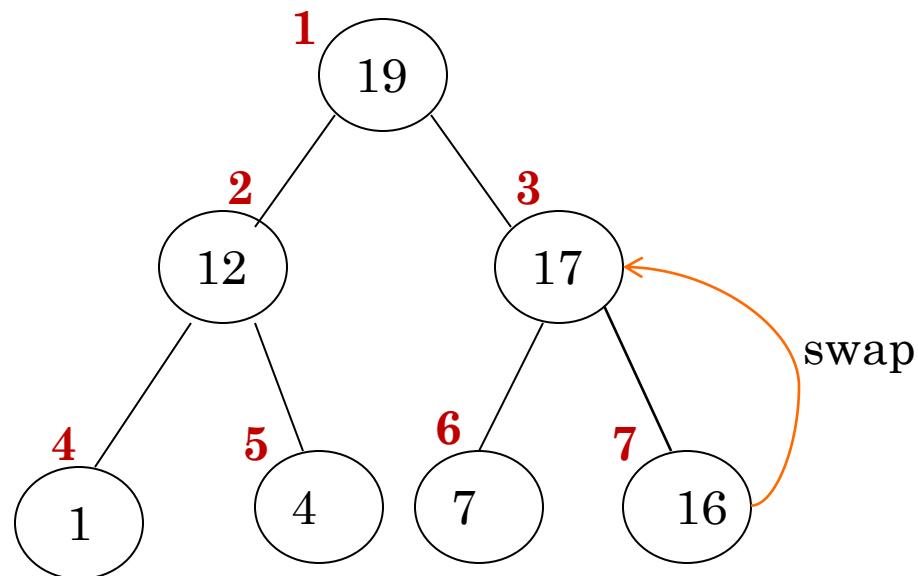
Insert 17

PARENT ( $i$ )  
return floor( $i/2$ )

19	12	16	1	4	7	17
1	2	3	4	5	6	7



19	12	17	1	4	7	16
1	2	3	4	5	6	7



Percolate up to maintain the  
heap property

**Data:**  $B$ : input array;  $N$ : starting index;  $newValue$ : new node

**Result:** Heap tree with the new node

**Procedure Insertion( $B$ ,  $N$ ,  $newValue$ )** // Array  $B$ , heap size  $N$ , new node  $newValue$

$N = N+1$ ; // Create a space in a heap to add the new node

$B[N] = newValue$ ;

$k = N$ ;

**while**  $k > 1$  **do**

$PNode = k/2$ ;       // **PNode** to denote the parent node

**if**  $B[PNode] < B[k]$  **then**

$swap(B[PNode], B[k])$ ;

$k = PNode$ ;

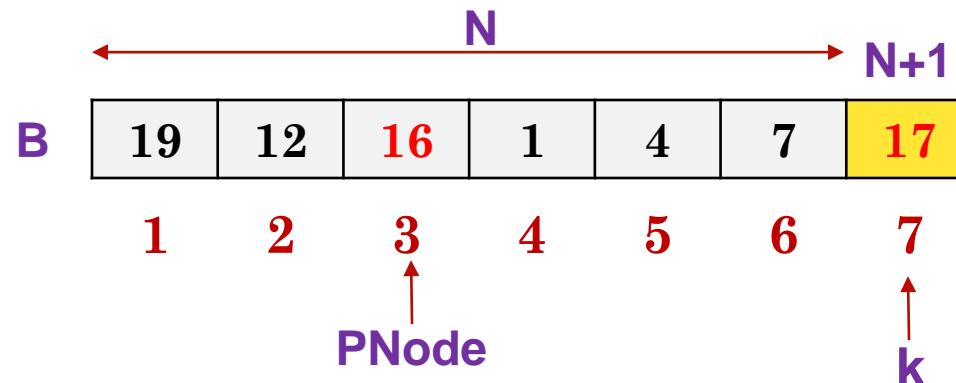
**else**

**return**;

**end**

**end**

**end**



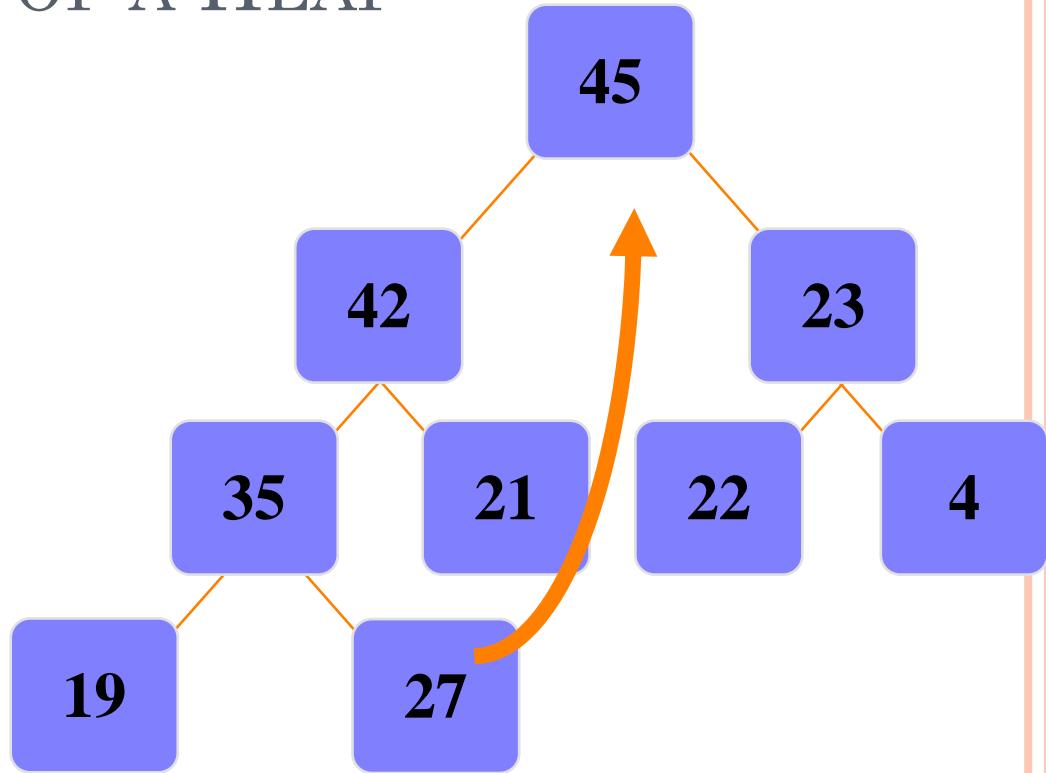
# DELETION

- Delete max
  - Copy the last number to the root ( overwrite the maximum element stored there ).
  - Restore the max heap property by percolate down.
- Delete min
  - Copy the last number to the root ( overwrite the minimum element stored there ).
  - Restore the min heap property by percolate down.



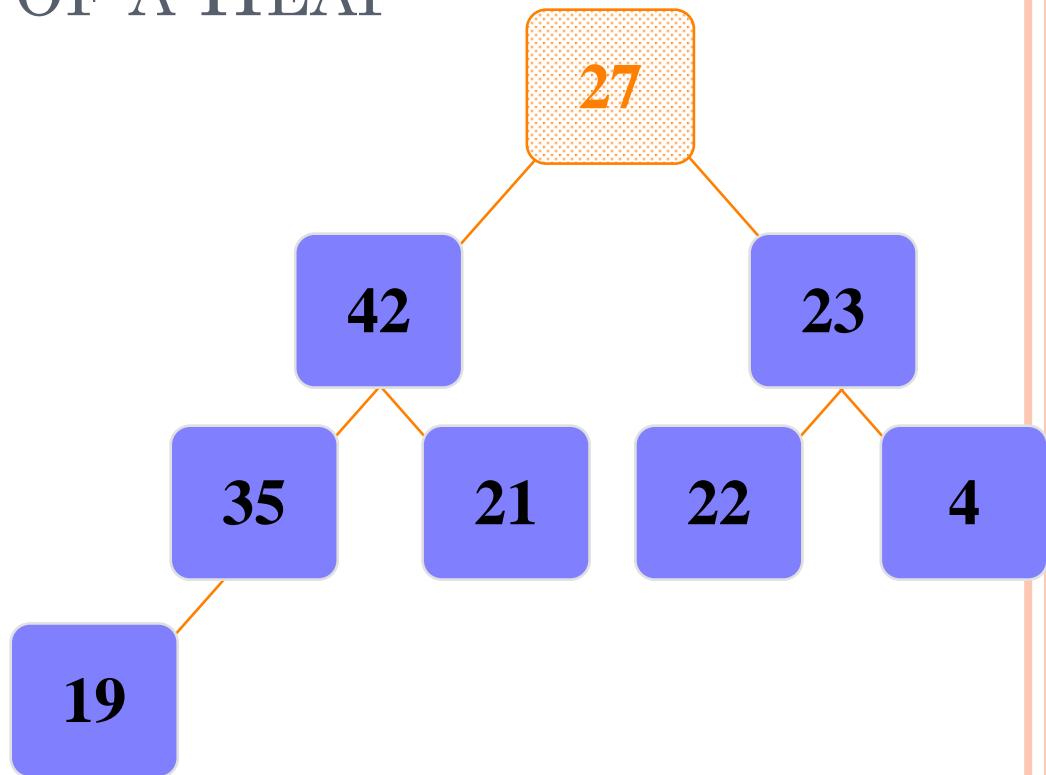
# MOVING THE TOP OF A HEAP

- Move the last node onto the root.



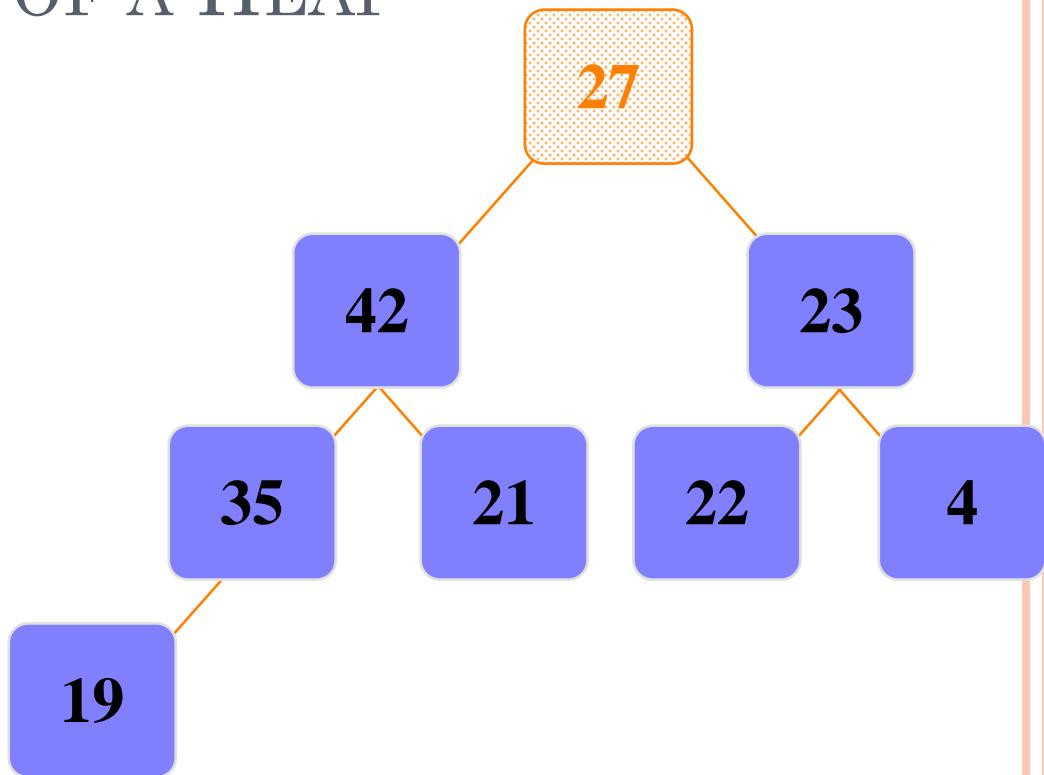
# MOVING THE TOP OF A HEAP

- Move the last node onto the root.



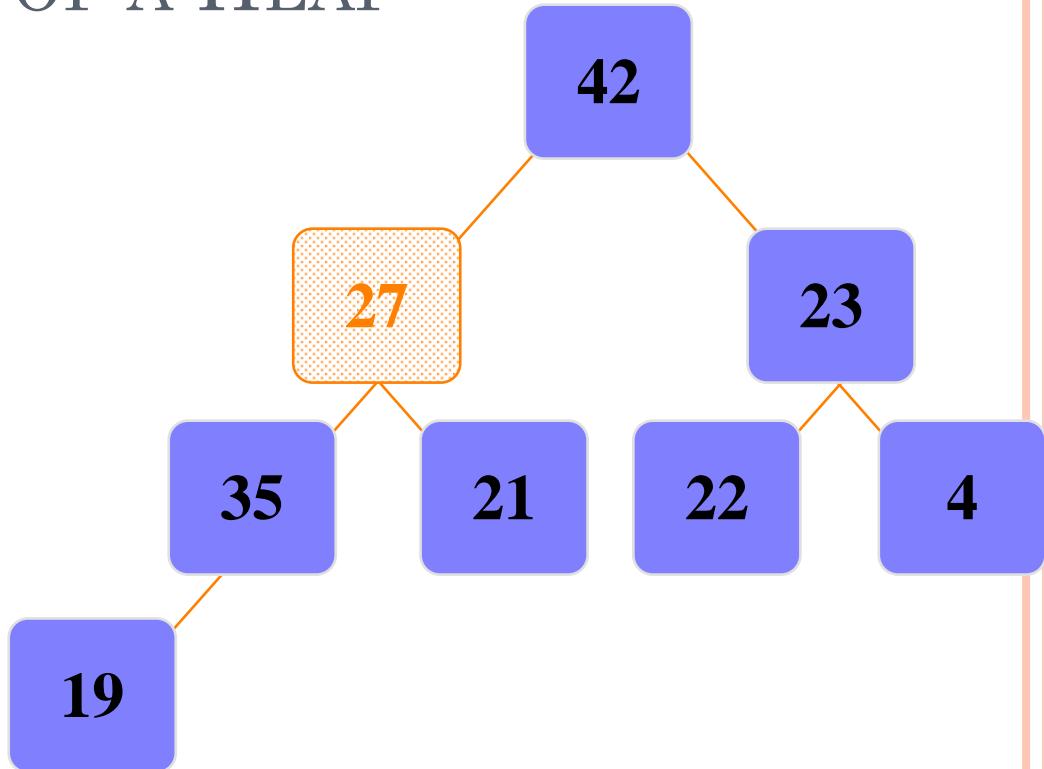
# MOVING THE TOP OF A HEAP

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



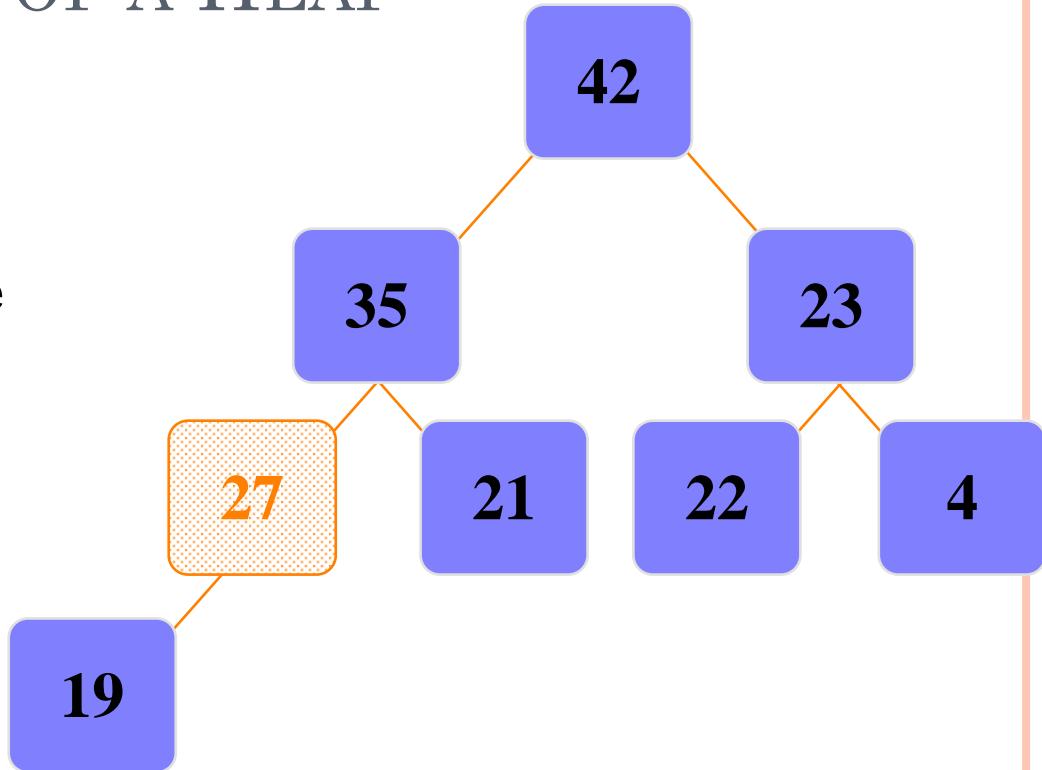
# MOVING THE TOP OF A HEAP

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



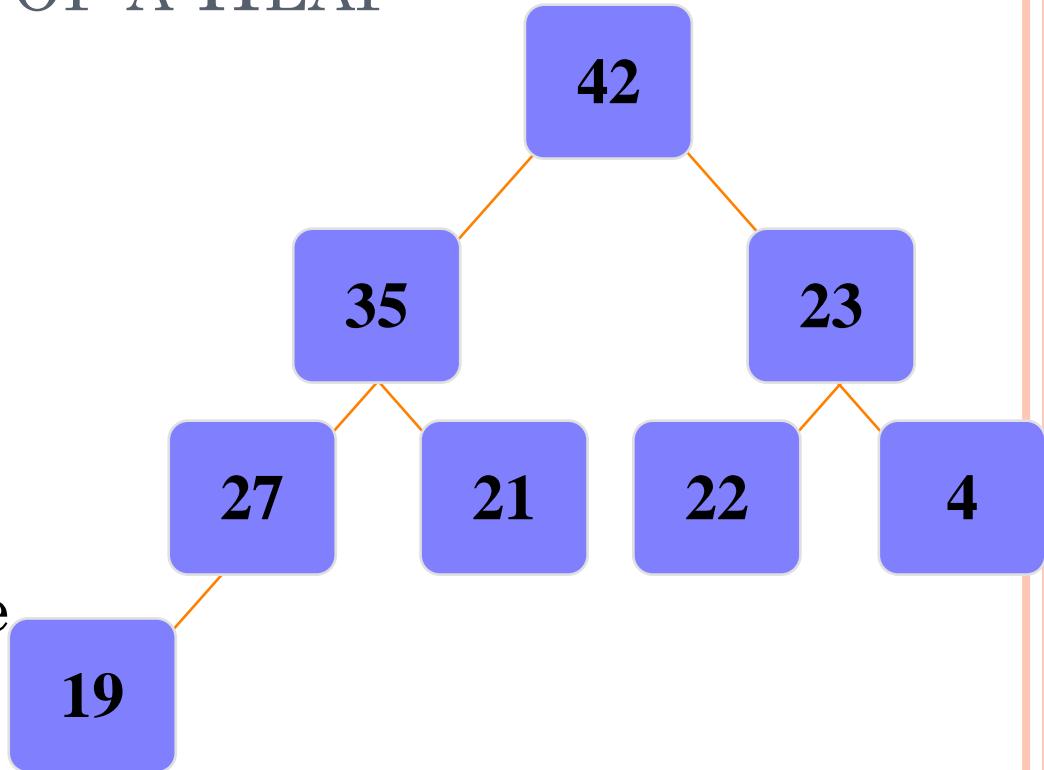
# MOVING THE TOP OF A HEAP

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.

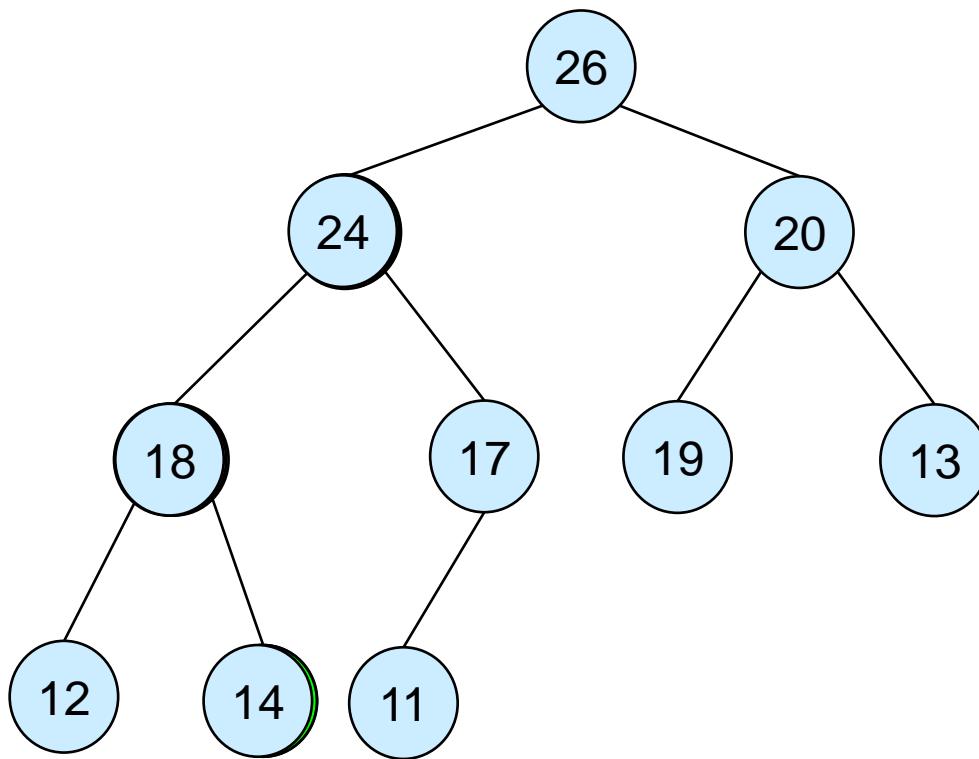


# MOVING THE TOP OF A HEAP

- ❑ The children all have keys  $\leq$  the out-of-place node, or
- ❑ The node reaches the leaf.
- ❑ The process of pushing the new node downward is called **reheapification downward**.



# MAXHEAPIFY – EXAMPLE



## HEAP SORT

- Heapsort is a popular and efficient sorting algorithm.
- The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.



# PROCEDURES ON HEAP SORT

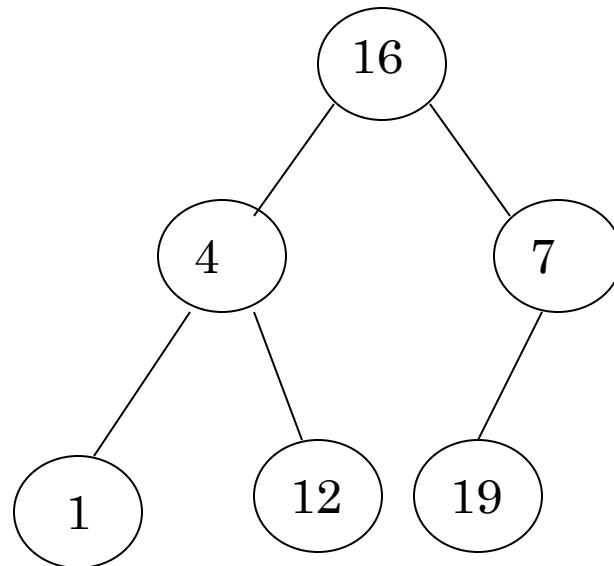
- In heap sorting, basically, there are two phases involved in the sorting of elements.
- By using the heap sort algorithm, they are as follows -
  - The first step includes the creation of a heap by adjusting the elements of the array.
  - After the heap creation, remove the heap's root element repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

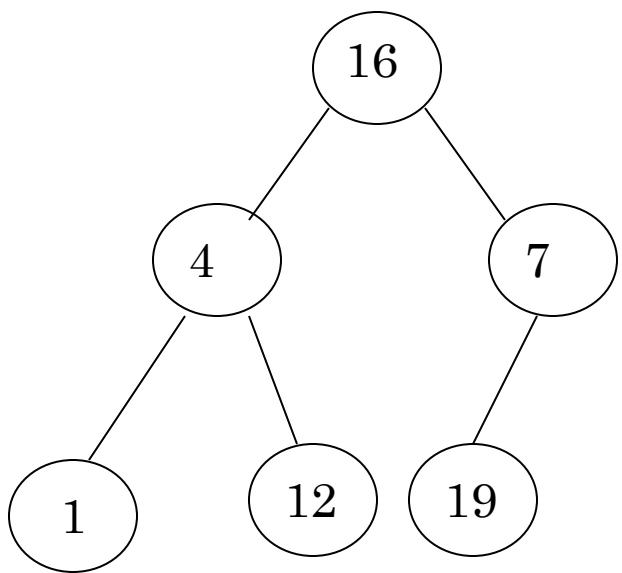


**Example:** Convert the following array to a heap

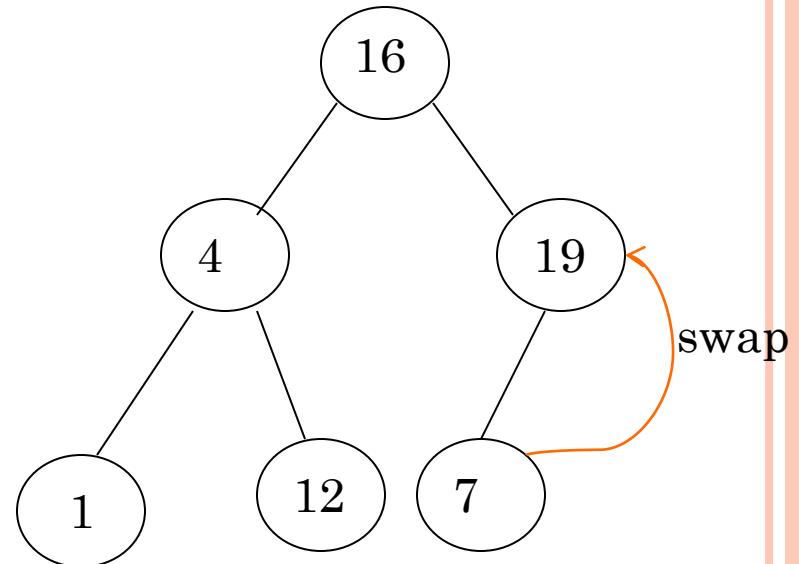
16	4	7	1	12	19
----	---	---	---	----	----

**Picture the array as a complete binary tree:**

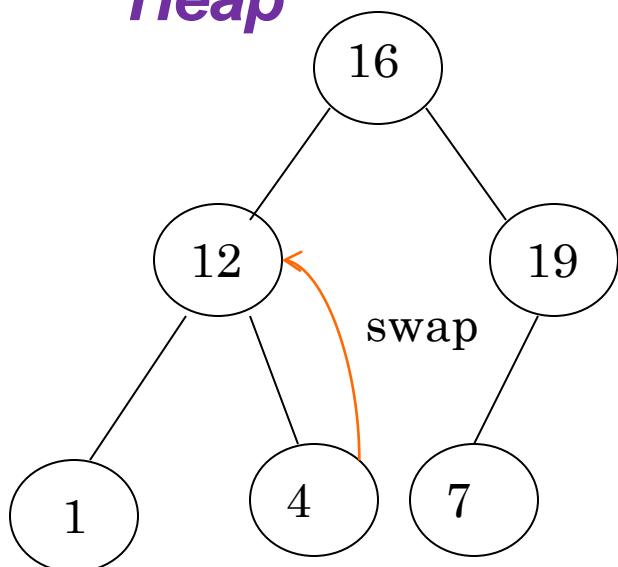




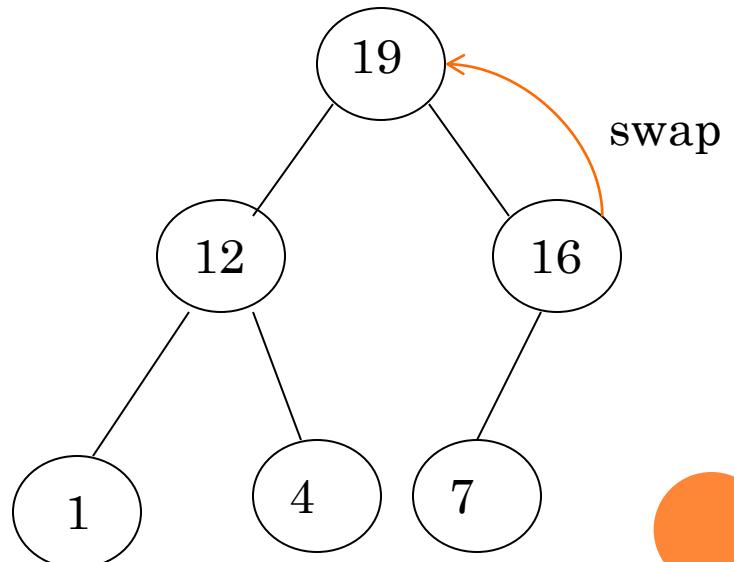
**Heapify**



**Heap**

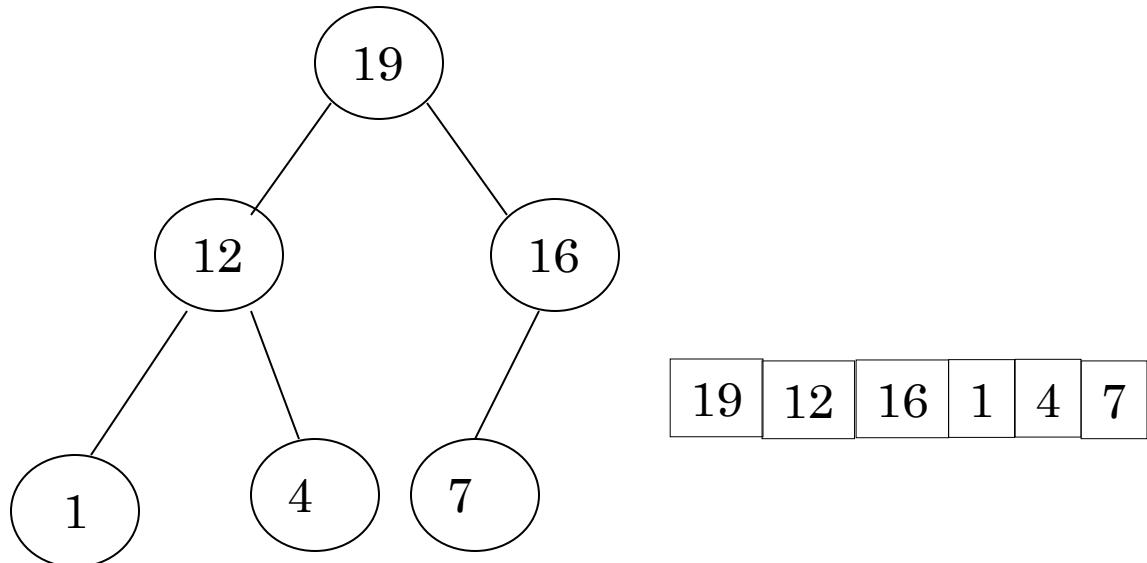


**Heapify**

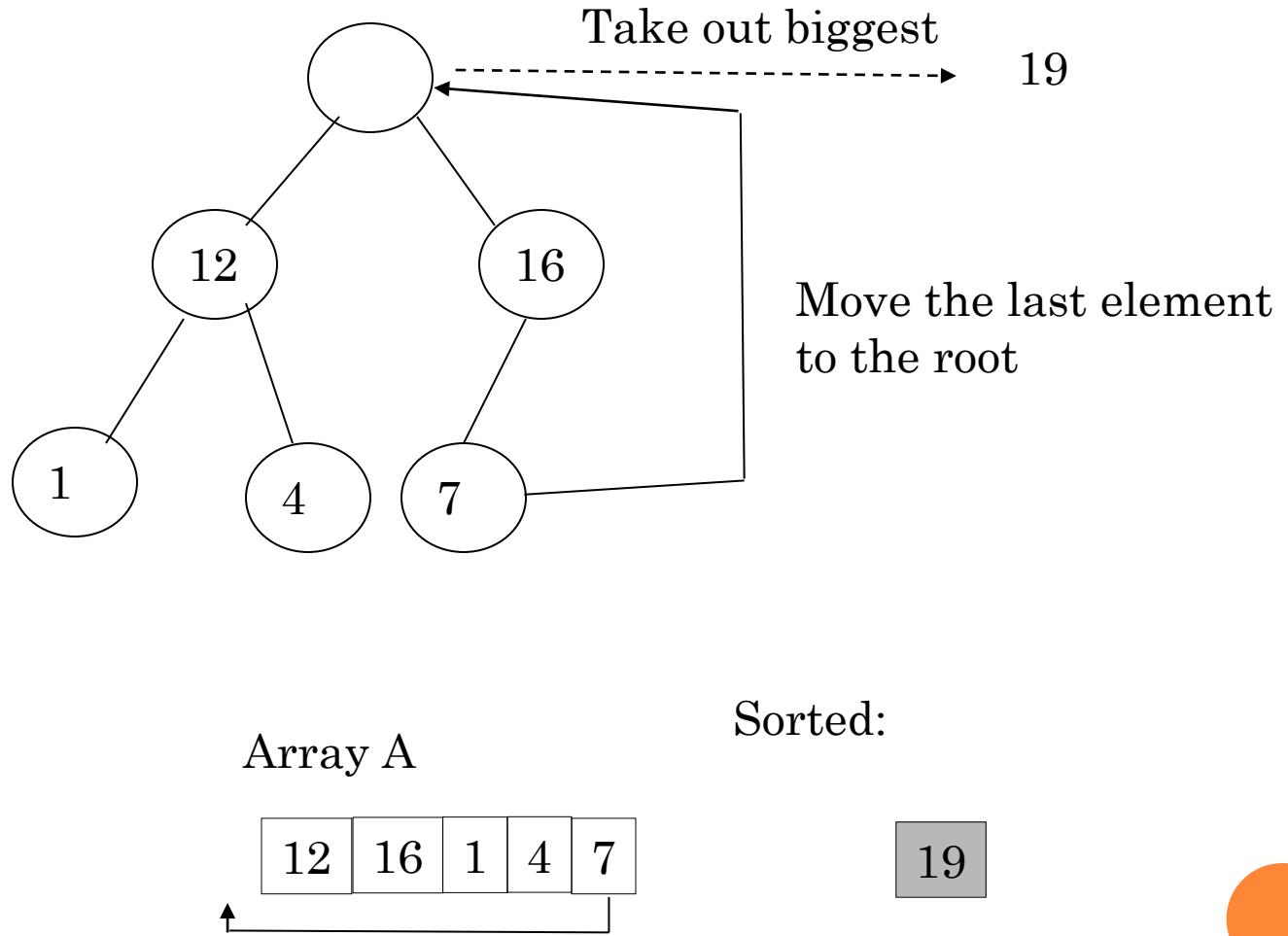


**Max Heap**

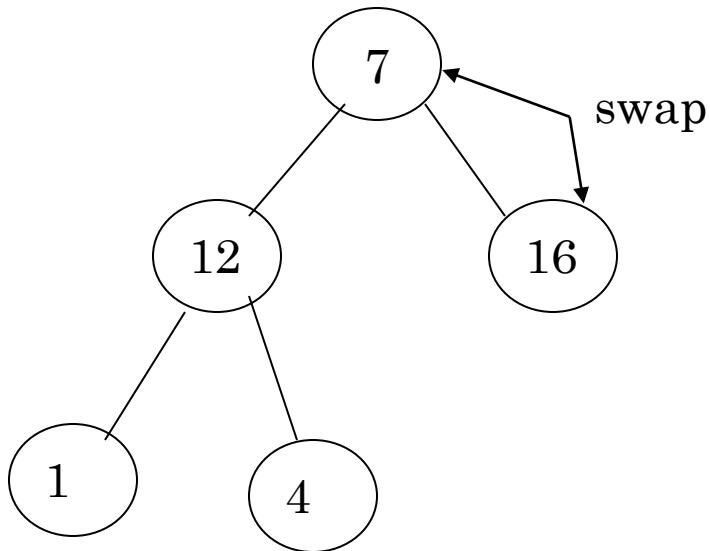
- To sort the elements in **increasing order**, use a **max heap**
- To sort the elements in **decreasing order**, use a **min heap**



# DELETE ELEMENTS FROM MAX HEAP



HEAPIFY()



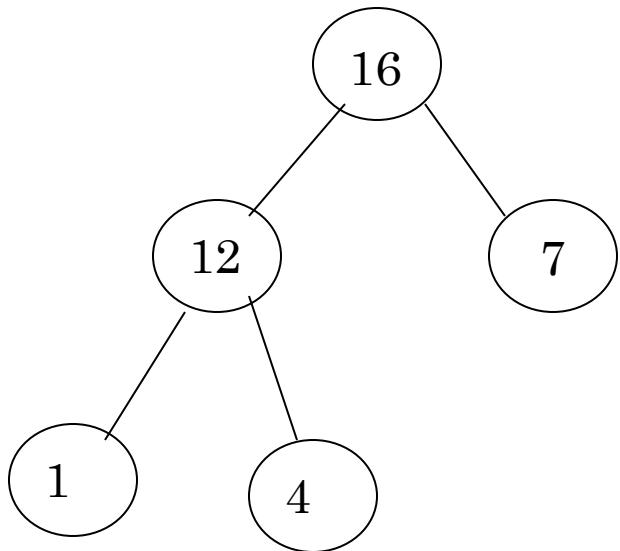
Array A

7	12	16	1	4
---	----	----	---	---

Sorted:

19
----





Array A

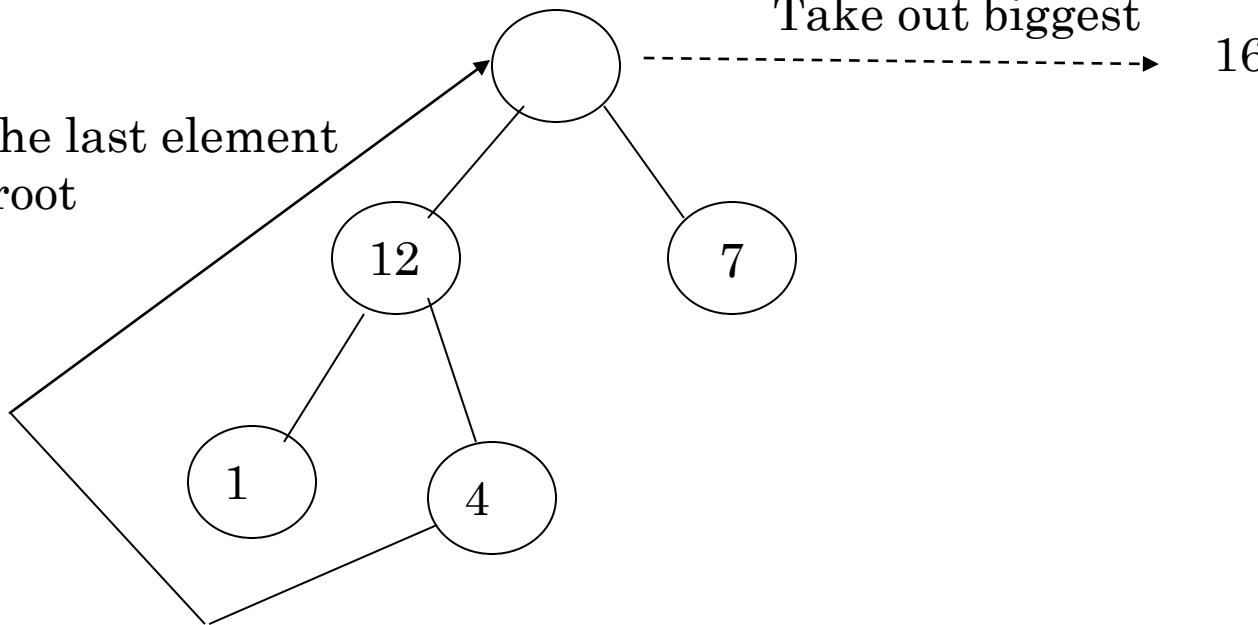
16	12	7	1	4
----	----	---	---	---

Sorted:

19



Move the last element  
to the root

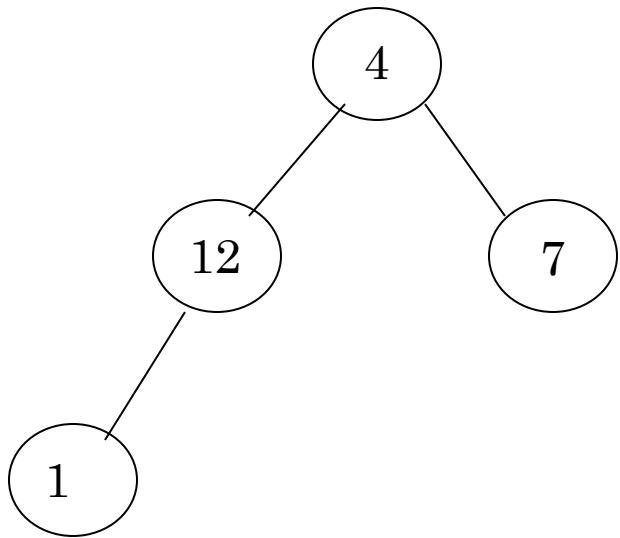


Array A



Sorted:





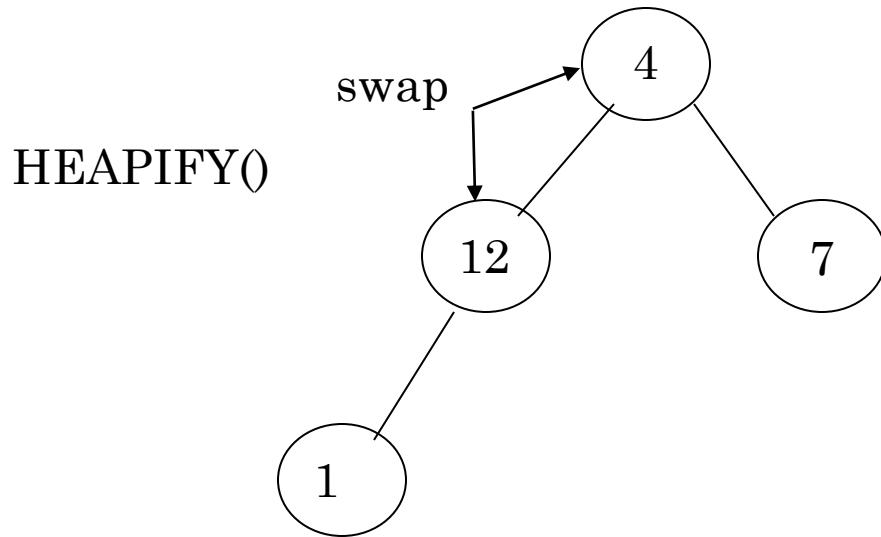
Array A

4	12	7	1
---	----	---	---

Sorted:

16	19
----	----





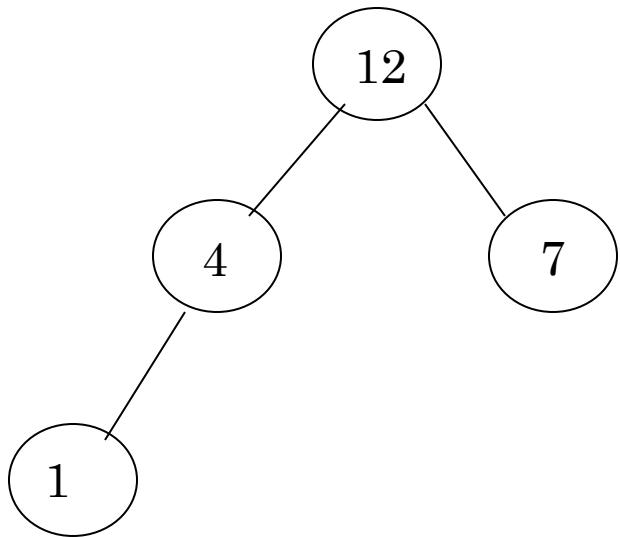
Array A

4	12	7	1
---	----	---	---

Sorted:

16	19
----	----





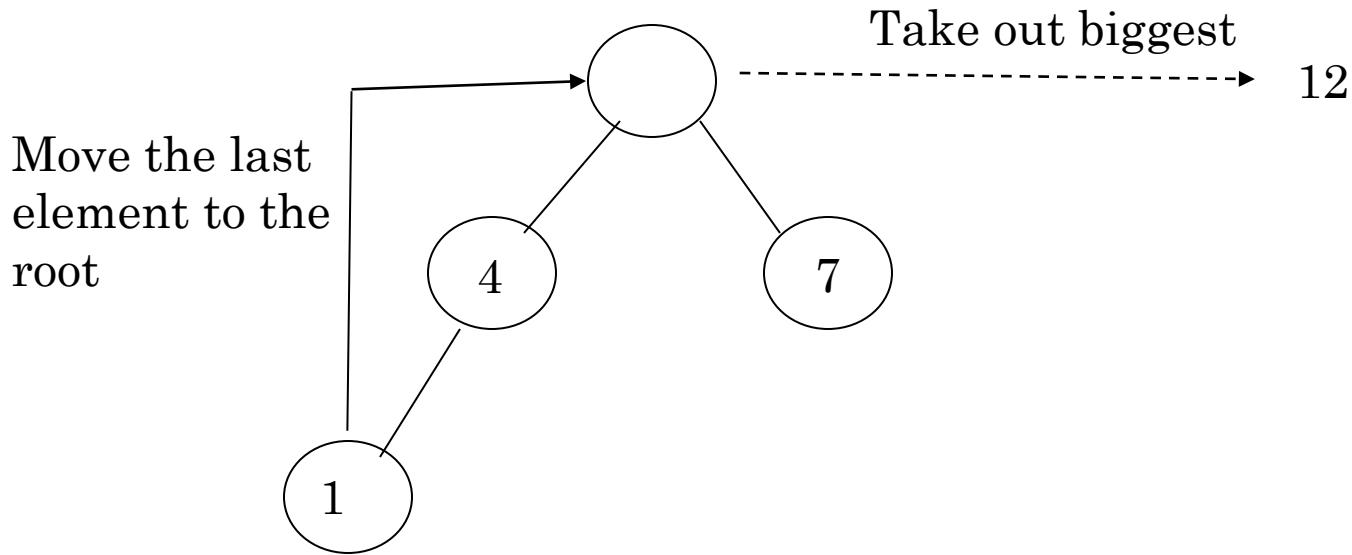
Array A

12	4	7	1
----	---	---	---

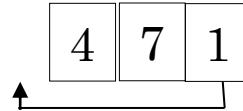
Sorted:

16	19
----	----



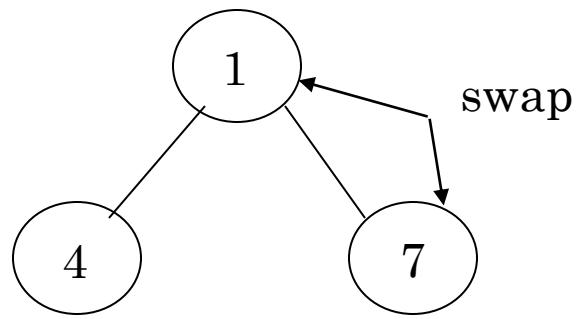


Array A



Sorted:





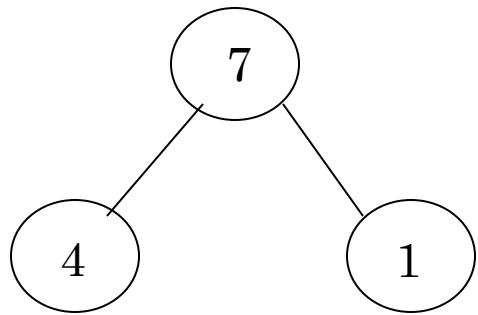
Array A

1	4	7
---	---	---

Sorted:

12	16	19
----	----	----





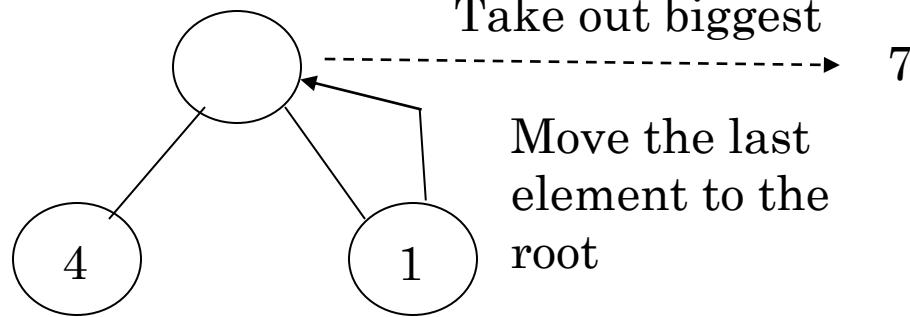
Array A

7	4	1
---	---	---

Sorted:

12	16	19
----	----	----





Array A

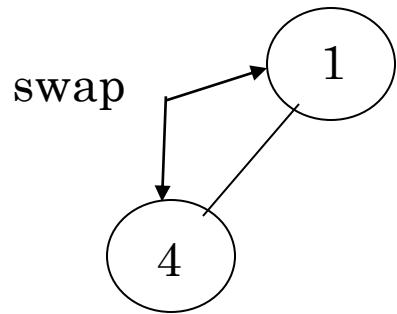
1	4
---	---

Sorted:

7	12	16	19
---	----	----	----



**HEAPIFY()**



Array A

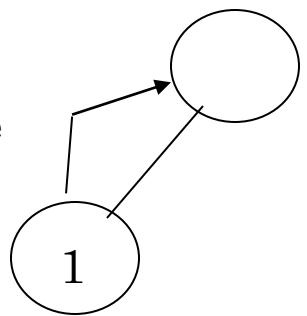
4	1
---	---

Sorted:

7	12	16	19
---	----	----	----



Move the last  
element to the  
root



Take out biggest

4

Array A



Sorted:



1

Take out biggest

Array A

Sorted:

1	4	7	12	16	19
---	---	---	----	----	----



Sorted:

1	4	7	12	16	19
---	---	---	----	----	----

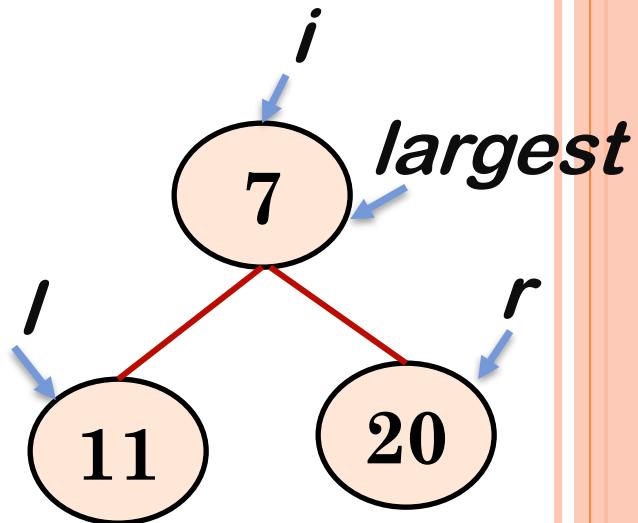


# HEAPIFY

- Heapify picks the largest child key and compares it to the parent key. If the parent key is larger then heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent now becomes larger than its children.

Heapify(A, n, i)

```
{  
    int largest=i;  
    int l < left(2*i);  
    int r < right(2*i)+1;  
    while(l <= n && A[l] > A[largest])  
    {  
        largest =l;  
    }  
    while(r <= n && A[r] > A[largest])  
    {  
        largest =r;  
    }  
    if (largest != i)  
    {  
        swap(A[largest],A[i])  
        Heapify(A, n, largest)  
    }  
}
```



```
Heapsort(A,n)
{
    for(i =n/2,i>=1,i--) //Build Max heap
    {
        Heapify(A,n, i);
    }
    for(i =n,i>=1,i--) //Deleting elements from Max heap
    {
        swap(A[1],A[i]);
        Heapify(A,n, 1)
    }
}
```

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

- The space complexity of the Heap sort is  $O(1)$

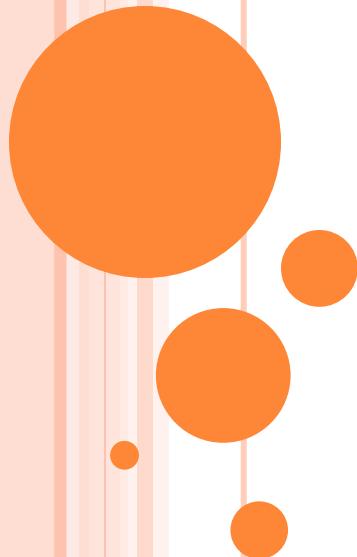
# POSSIBLE APPLICATIONS

- When we want to know the task that carry the highest priority given a large number of things to do
- Interval scheduling, when we have a list of certain tasks with start and finish times and we want to do as many tasks as possible
- Implementation of priority
- Sorting a list of elements that need an efficient sorting algorithm



# PRIORITY QUEUE USING HEAPS

Dr.Priyanka N



## PRIORITY QUEUE

- The priority queue in the data structure is an extension of the “normal” queue
- It is an abstract data type that contains a group of items
- It is like the “normal” queue except that the **dequeuing** elements follow a priority order



# QUEUE

- The “normal” queue follows a pattern of **First-In-First-Out (FIFO)**
- It dequeues elements in the same order followed at the time of insertion operation



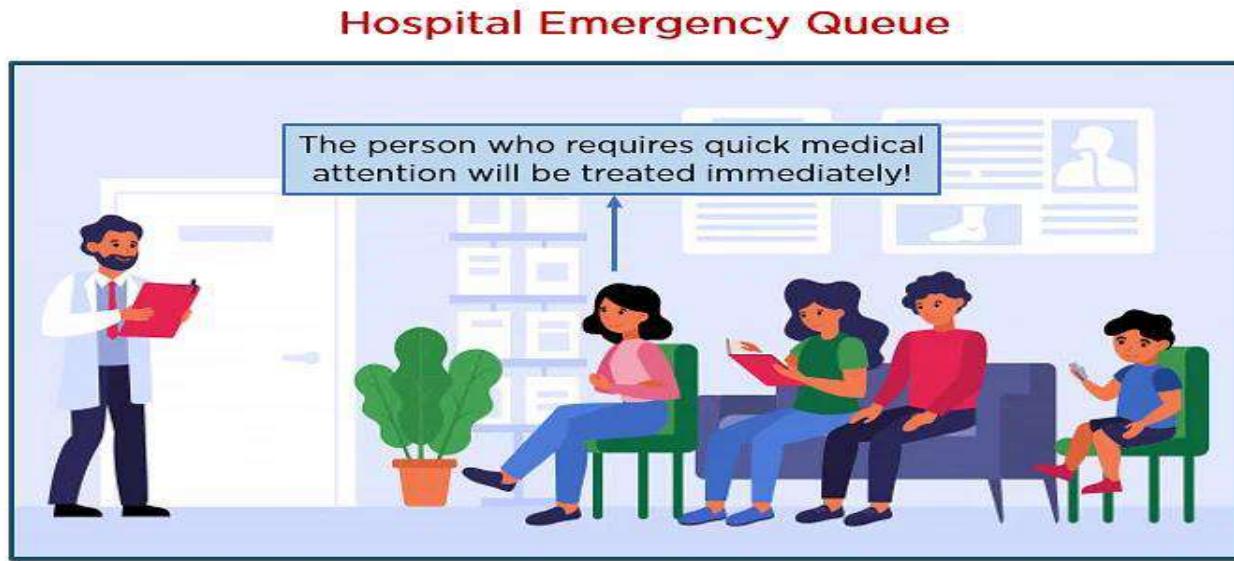
# PRIORITY QUEUE

- The element order in a priority queue depends on the **element's priority** in that queue
- The priority queue moves the highest priority elements at the beginning of the priority queue and the lowest priority elements at the back of the priority queue
- It supports only those elements that are comparable
- Hence, a priority queue in the data structure arranges the elements in either **ascending** or **descending** order



# EXAMPLE

- Think of a priority queue as several patients waiting in line at a hospital
- Here, the situation of the patient defines the priority order
- The patient with the most severe injury would be the first in the queue



# IMPLEMENTATION OF THE PRIORITY QUEUE IN DATA STRUCTURE

- Linked list
- Binary heap
- Arrays
- Binary search tree

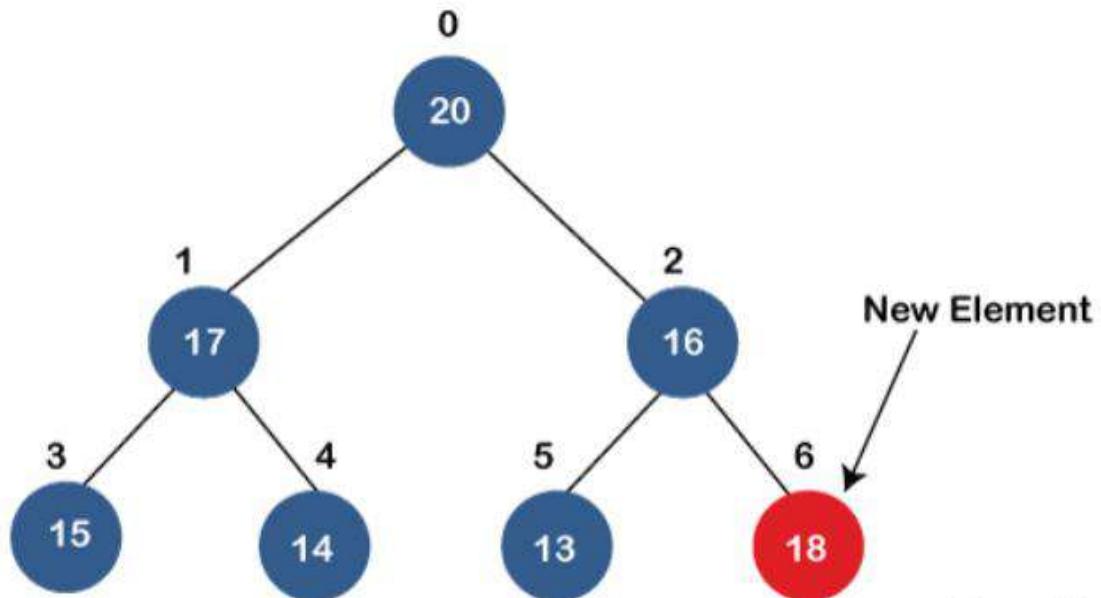


# CHARACTERISTICS OF A PRIORITY QUEUE

- A queue is termed as a priority queue if it has the following characteristics:
  - Each item has some priority associated with it
  - An item with the highest priority is moved to the front and deleted first
  - If two elements share the same priority value, then the priority queue follows the **First-In-First-Out** principle for de-queue operation

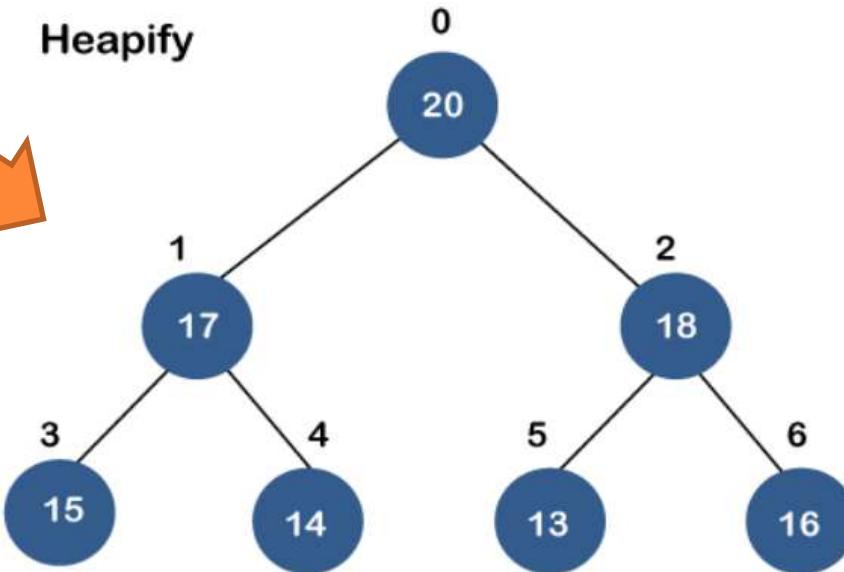


- o Inserting the element in a priority queue (max heap)

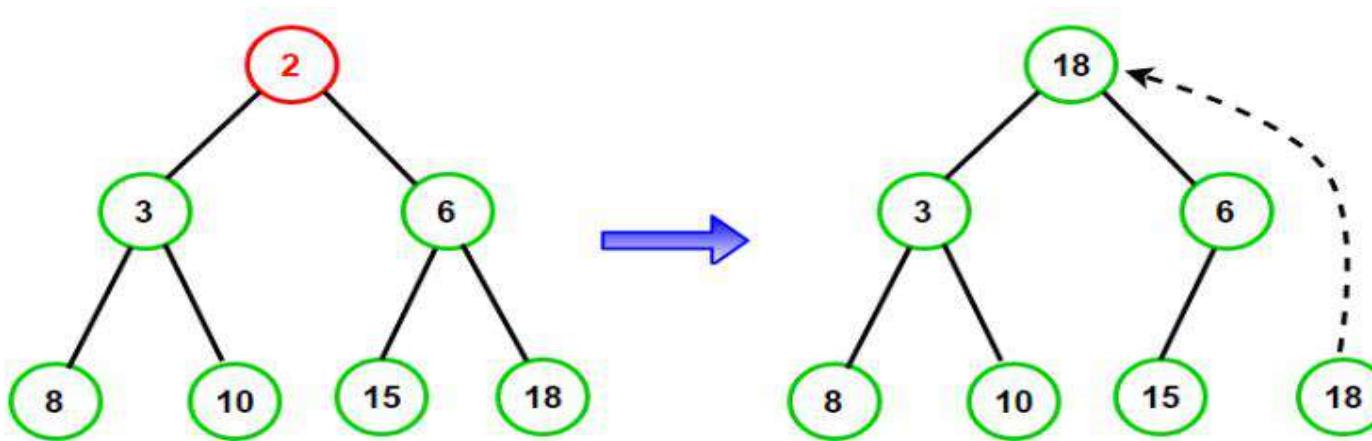


New Element

Heapify

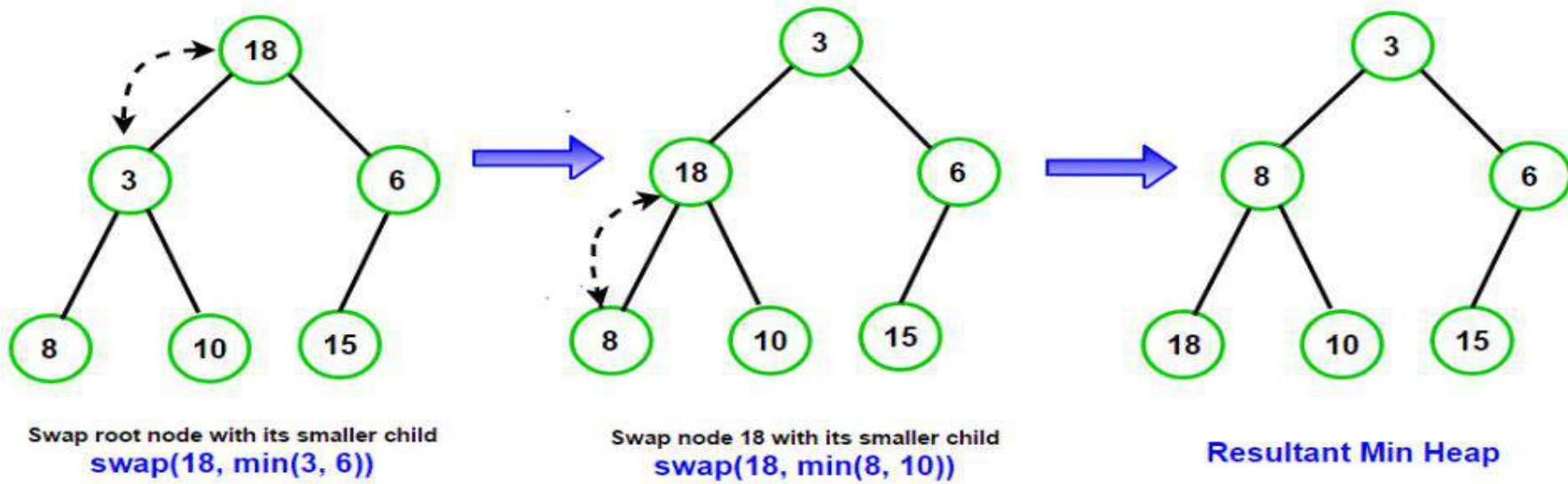


# REMOVING THE MINIMUM ELEMENT FROM THE PRIORITY QUEUE



**Pop() called on min heap**

Replace the root of the heap with the last element on the last level and call  
**Heapify-down(root)**



## **TYPES OF PRIORITY QUEUE**

- A priority queue is of two types:
  - Ascending Order Priority Queue
  - Descending Order Priority Queue



# ASCENDING ORDER PRIORITY QUEUE

- An ascending order priority queue gives the highest priority to the lower number in that queue

## Example

4	8	12	45	35	20
---	---	----	----	----	----

- Six numbers in the priority queue are 4, 8, 12, 45, 35, 20



## ASCENDING ORDER PRIORITY QUEUE

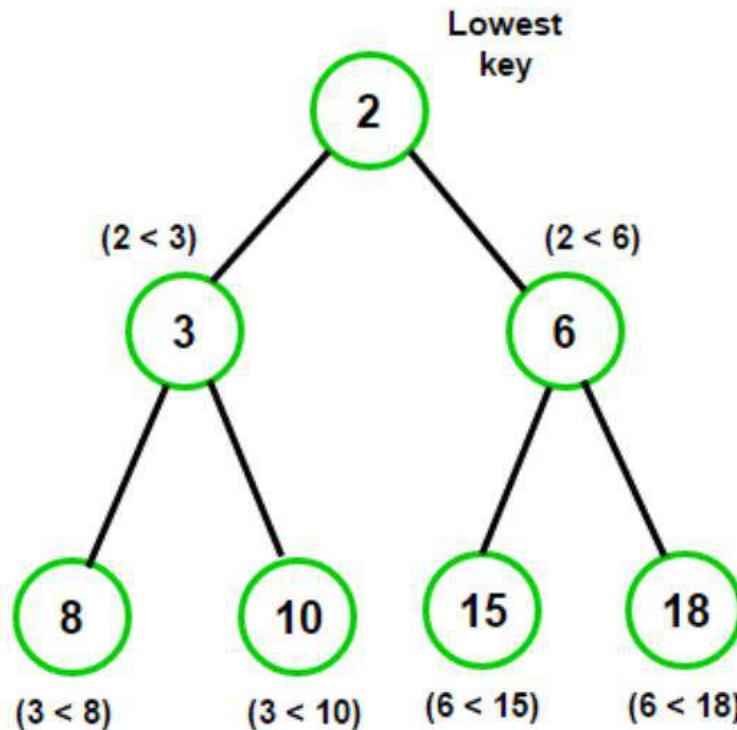
- Firstly, arrange the numbers in ascending order
- The new list is as follows:

4	8	12	20	35	45
---	---	----	----	----	----

- Here, **4** has the highest priority, and **45** has the lowest priority



# IMPLEMENTATION OF THE PRIORITY QUEUE USING BINARY HEAP



## Min Heap

(Parent key is less than or equal to ( $\leq$ ) the child key)

- The minimum element of a priority queue is at the root of the max-heap
- Return the element at the root of the heap

## DESCENDING ORDER PRIORITY QUEUE

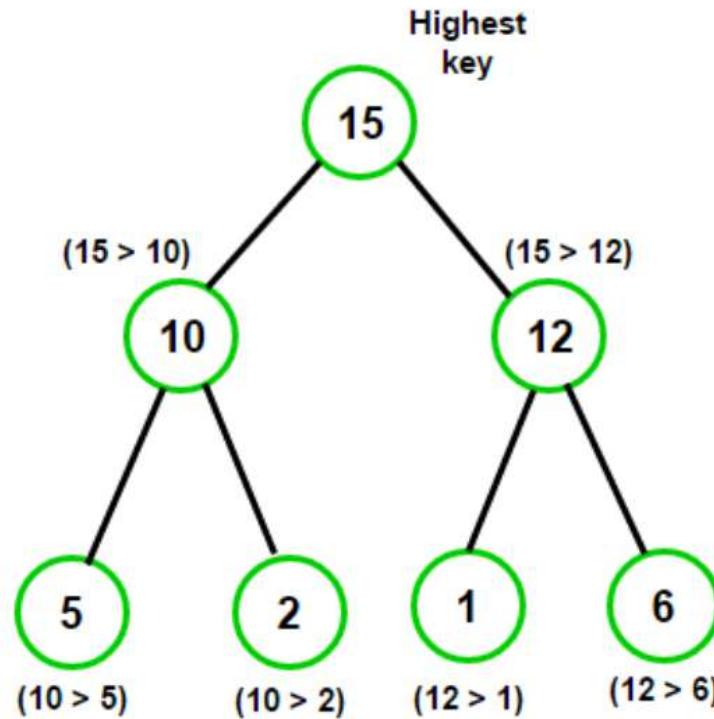
- A descending-order priority queue gives the highest priority to the highest number in that queue
- First, arrange the numbers in descending order
- The new list is as follows:

45	35	20	12	8	4
----	----	----	----	---	---

- Here, **45** has the highest priority, and **4** has the lowest priority



# IMPLEMENTATION OF THE PRIORITY QUEUE USING BINARY HEAP



**Max Heap**  
(Parent key is greater than or equal to ( $\geq$ ) the child key)

- The maximum element of a priority queue is at the root of the max-heap
- Return the element at the root of the heap

Operation	Unordered Array	Ordered Array	Binary Heap	Binary Search Tree
Insert	O(1)	O(N)	O(log(N))	O(log(N))
Peek	O(N)	O(1)	O(1)	O(1)
Delete	O(N)	O(1)	O(log (N))	O(log(N))



# **BINARY TREE**

---

**Dr.Priyanka N  
Assistant Professor Senior Grade 1  
SCOPE  
VIT University - Vellore**

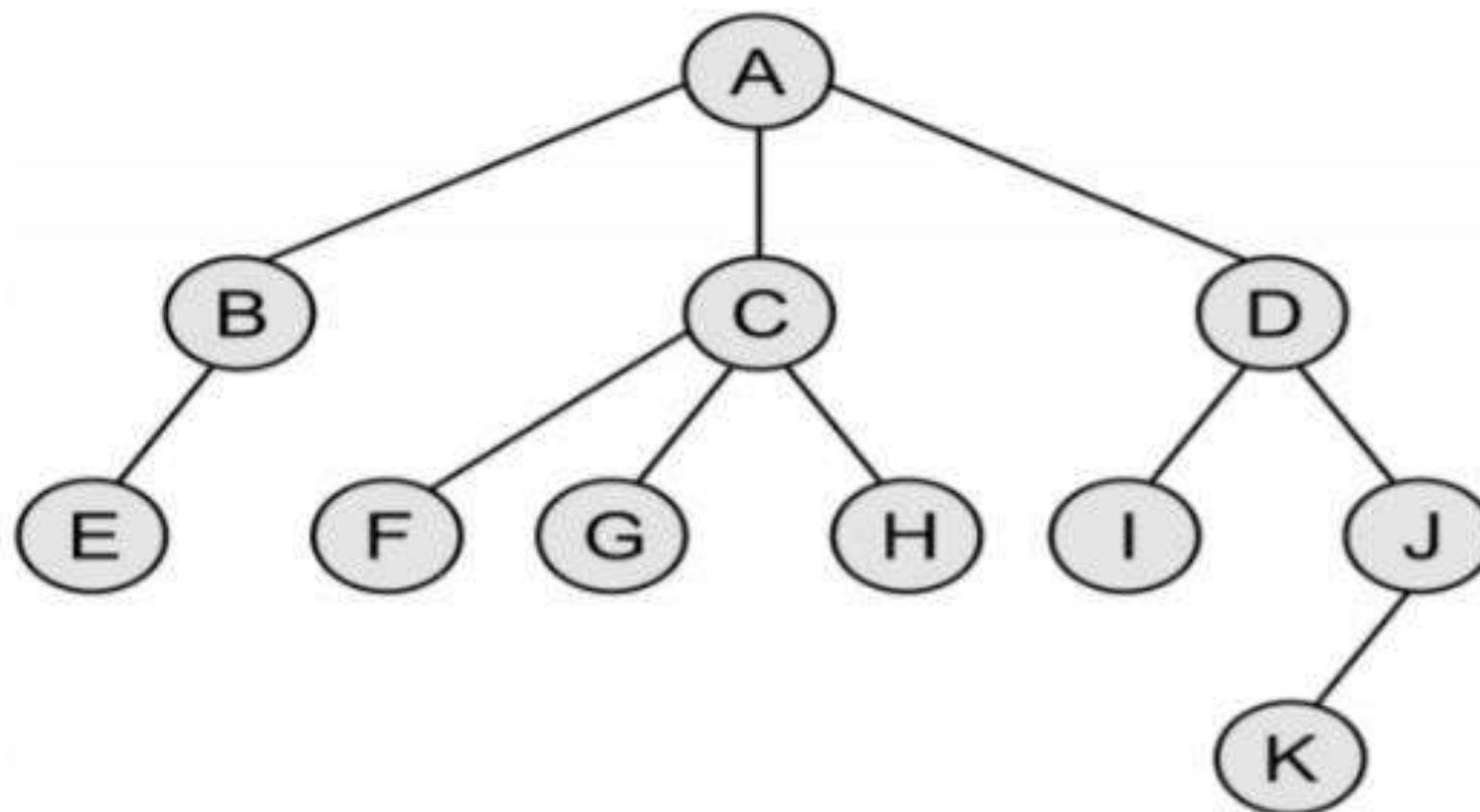
# CONVERT GENERAL TREE TO BINARY TREE

---

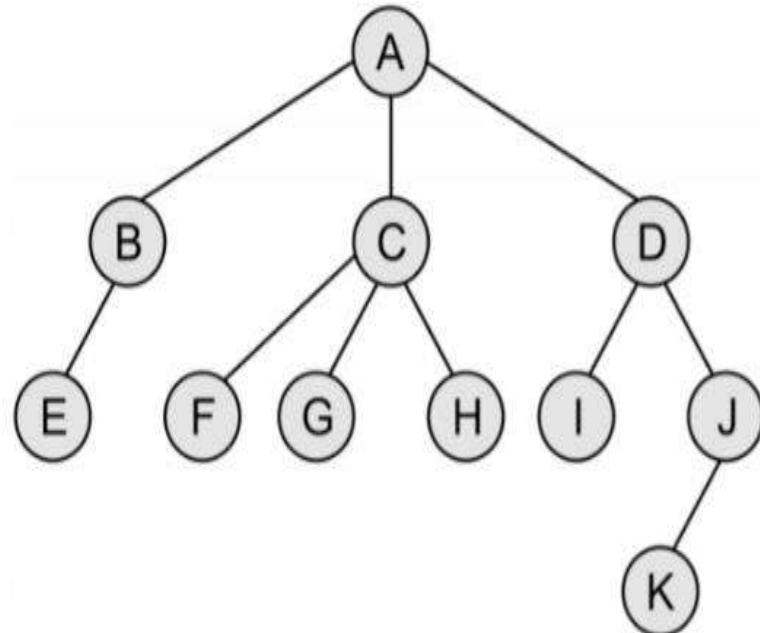
# Rules

- The root of the Binary Tree is the Root of the Generic Tree
- The left child of a node in the Generic Tree is the Left child of that node in the Binary Tree
- The right sibling of any node in the Generic Tree is the Right child of that node in the Binary Tree

# Generic Tree (N-array Tree)



# Binary Tree



Step 1

Step 2

Step 3

Step 4

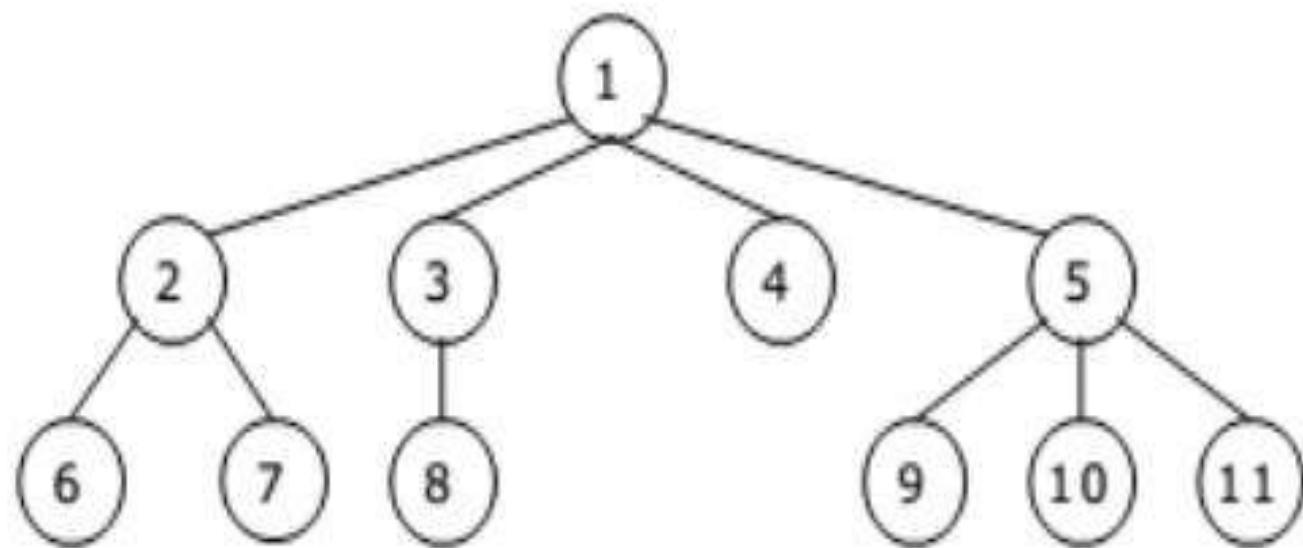
Step 5

Step 6

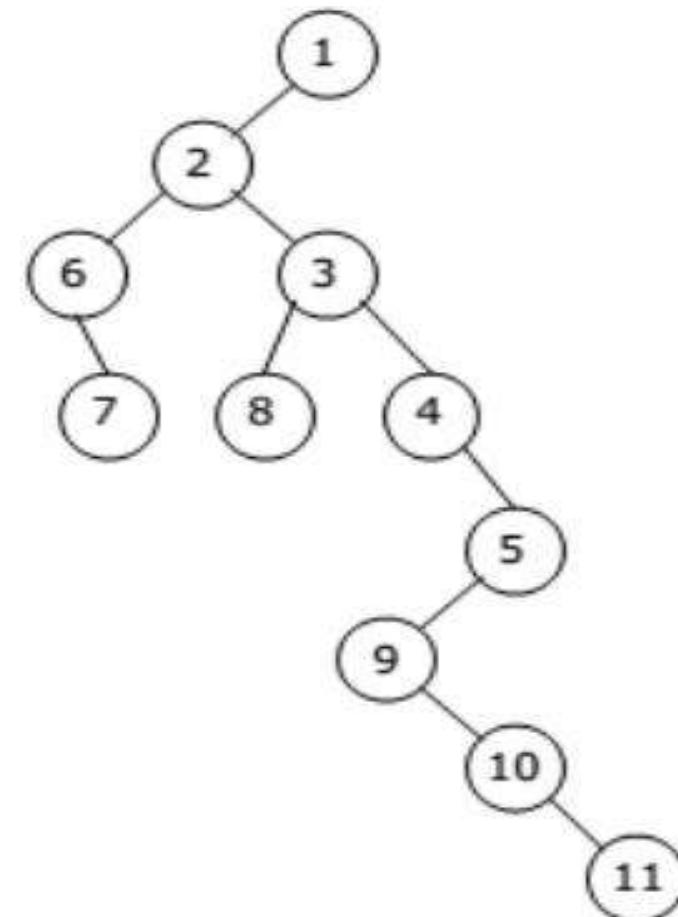
Step 7

Step 8

# Generic Tree(N-array Tree)



# Binary Tree



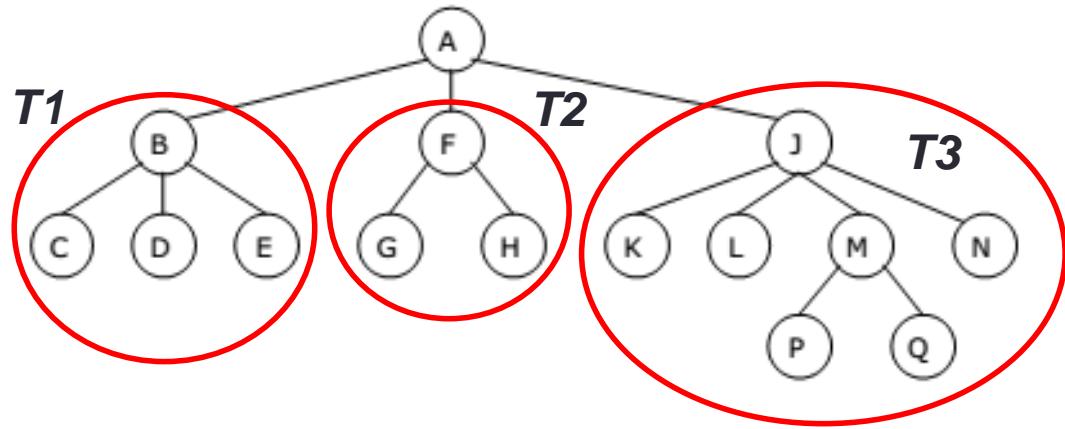
# GENERAL TRAVERSALS

---

# RULES

- Preorder:
  - 1) Process the root R.
  - 2) Traverse the subtree T<sub>1</sub>, T<sub>2</sub>, ……, T<sub>M</sub> in preorder.
- Postorder:
  - 1) Traverse the subtree T<sub>1</sub>, T<sub>2</sub>, ……, T<sub>M</sub> in postorder.
  - 2) Process the root R.

# Preorder Traversal



The tree T has the root A and subtrees T1, T2 and T3 such that:  
T1 consists of nodes B, C, D and E.  
T2 consists of nodes F, G and H.  
T3 consists of nodes J, K, L, M, N, P and Q.

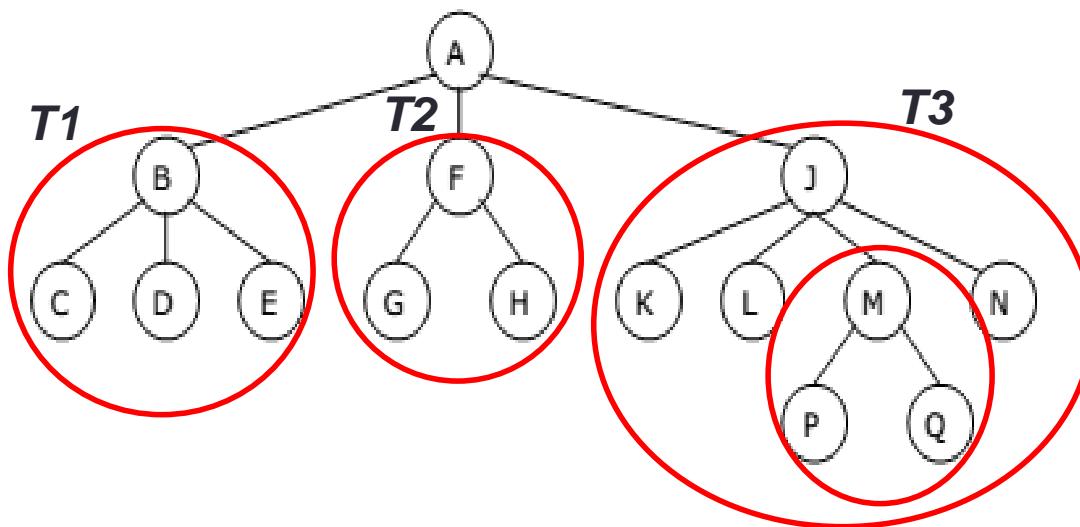
## STEPS:

- (i) Process root A.
- (ii) Traverse T1 in preorder: Process nodes B, C, D, E.
- (iii) Traverse T2 in preorder: Process nodes F, G, H.
- (iv) Traverse T3 in preorder: Process nodes J, K, L, M, P, Q, N.

The preorder traversal of T is as follows:

A, B, C, D, E, F, G, H, J, K, L, M, P, Q, N

# Postorder Traversal



The tree T has the root A and subtrees T1, T2 and T3 such that:  
T1 consists of nodes B, C, D and E.  
T2 consists of nodes F, G and H.  
T3 consists of nodes J, K, L, M, N, P and Q.

## STEPS:

- (i) Traverse T1 in postorder: Process nodes C, D, E, B.
- (ii) Traverse T2 in postorder: Process nodes G, H, F.
- (iii) Traverse T3 in postorder: Process nodes K, L, P, Q, M, N, J.
- (iv) Process root A.

The postorder traversal of T is as follows:

C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A

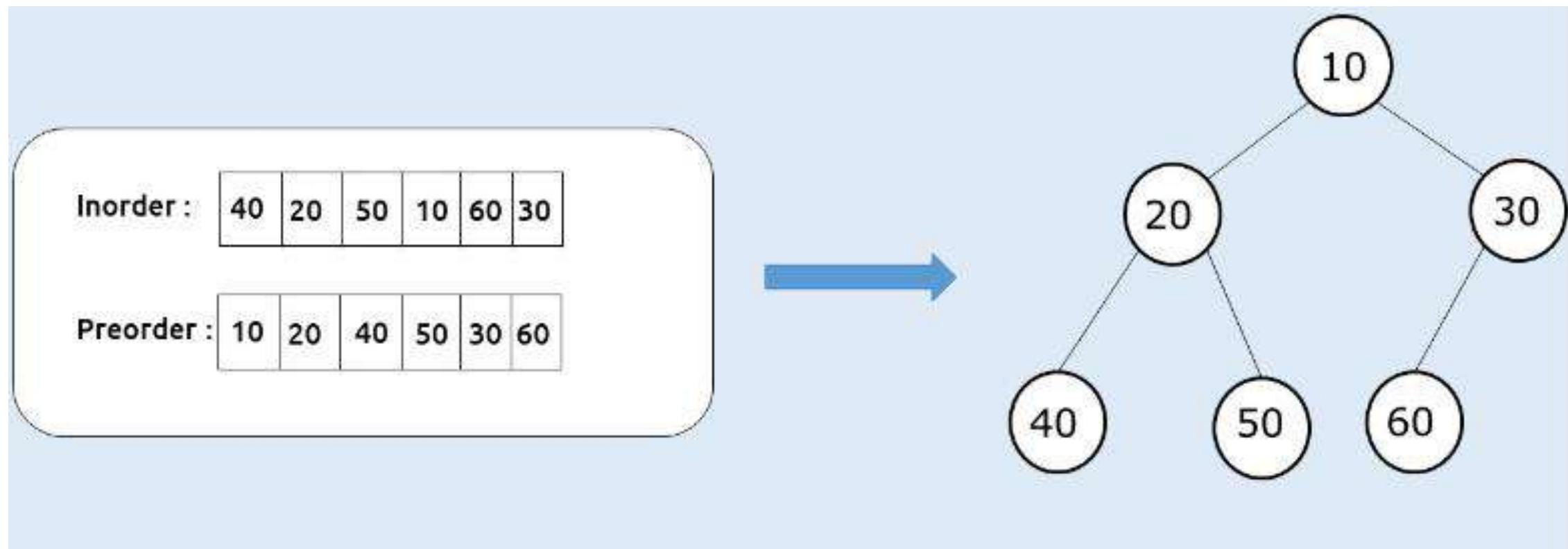
# Inorder Traversal

- Inorder traversal **does not have a natural definition** for the general tree, because there is no particular number of children for an internal node
- **Method 1:** Visit the leftmost subtree in inorder, then the root, then visit the remaining subtrees in inorder
- **Method 2:** Visit all the children except the last then the root and finally the last child recursively

# CONSTRUCT BINARY TREE FROM PREORDER AND INORDER

---

# Example:



# STEPS

- Inorder traversal is a special traversal that helps us to identify a node and its left and right subtree
- Preorder traversal always gives us the root node as the first element

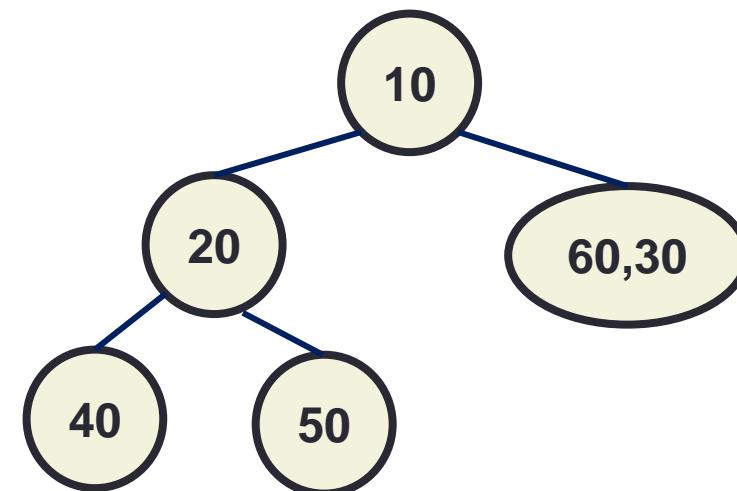
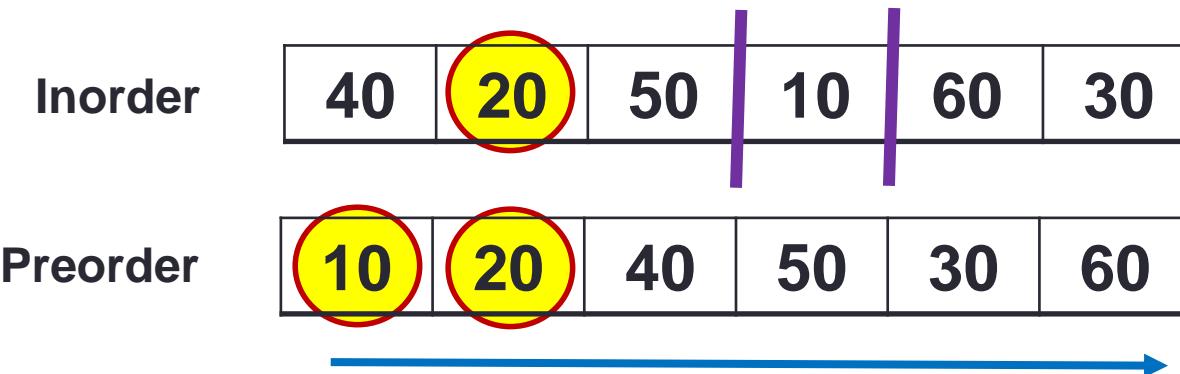
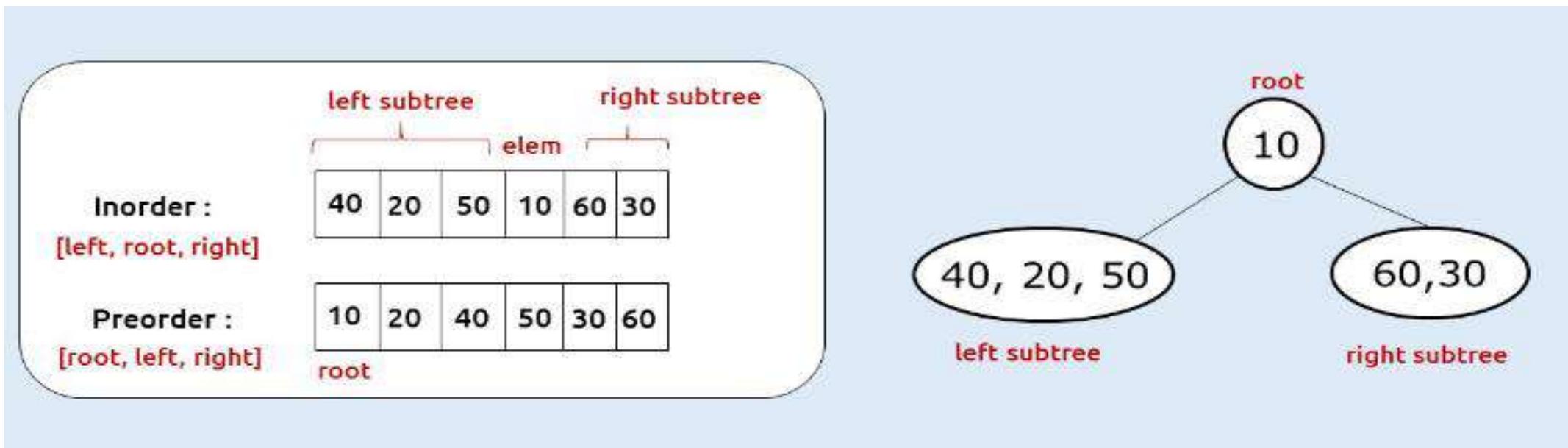
Inorder :	40   20   50   10   60   30
[left, root, right]	
Preorder :	10   20   40   50   30   60
[root, left, right]	

# STEPS

- Inorder traversal is a special traversal that helps us to identify a node and its left and right subtree
- Preorder traversal always gives us the root node as the first element

Inorder :	40   20   50   10   60   30
[left, root, right]	
Preorder :	10   20   40   50   30   60
[root, left, right]	

# STEPS

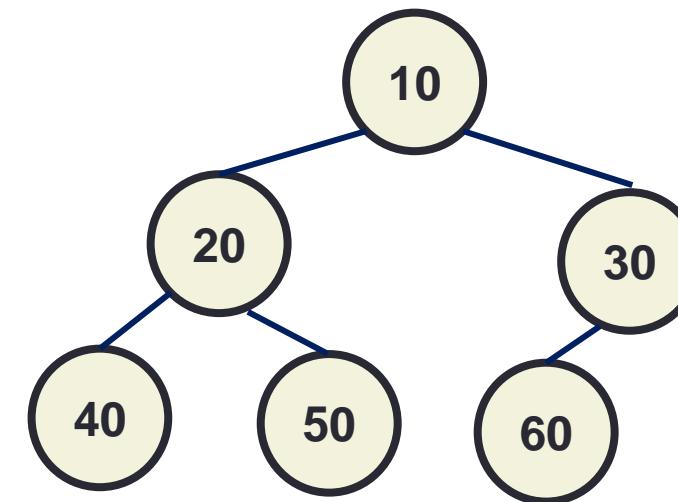


# STEPS

Inorder



Preorder



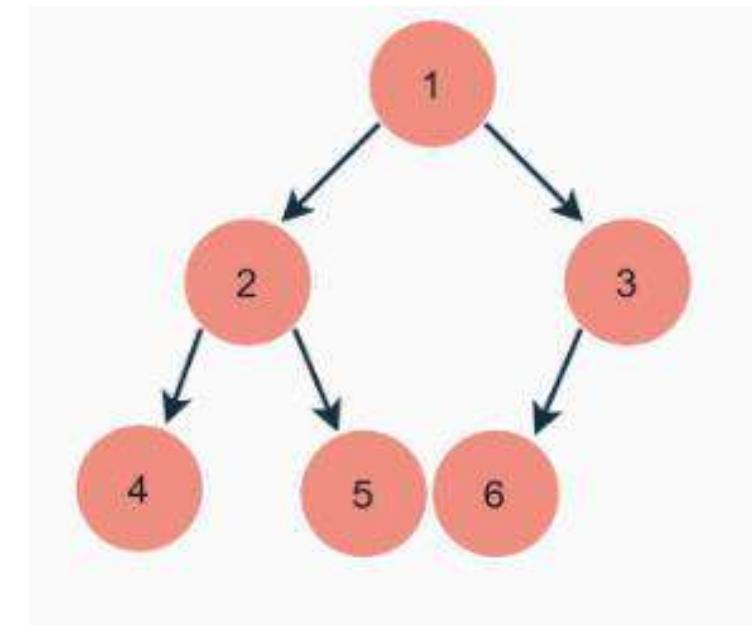
# Try it on your own

Inorder

4	2	5	1	6	3
---	---	---	---	---	---

Preorder

1	2	4	5	3	6
---	---	---	---	---	---



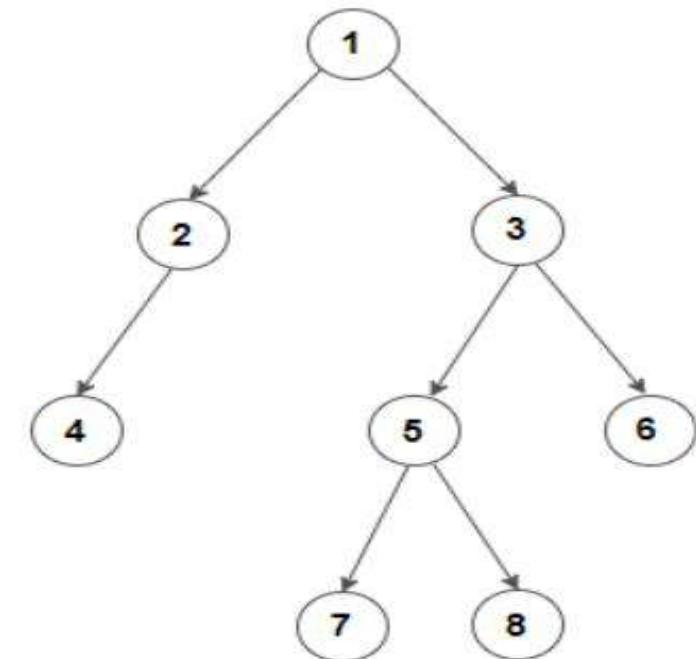
# CONSTRUCT BINARY TREE FROM POSTORDER AND INORDER

---

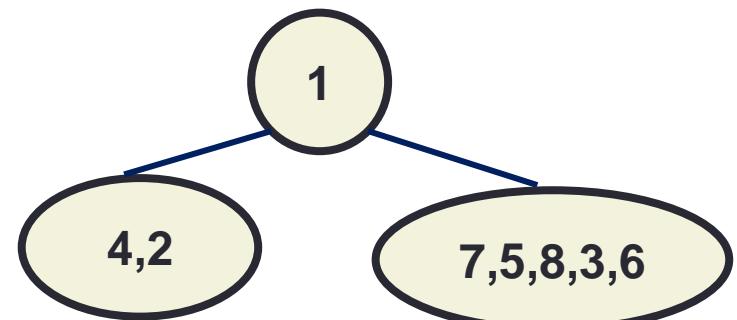
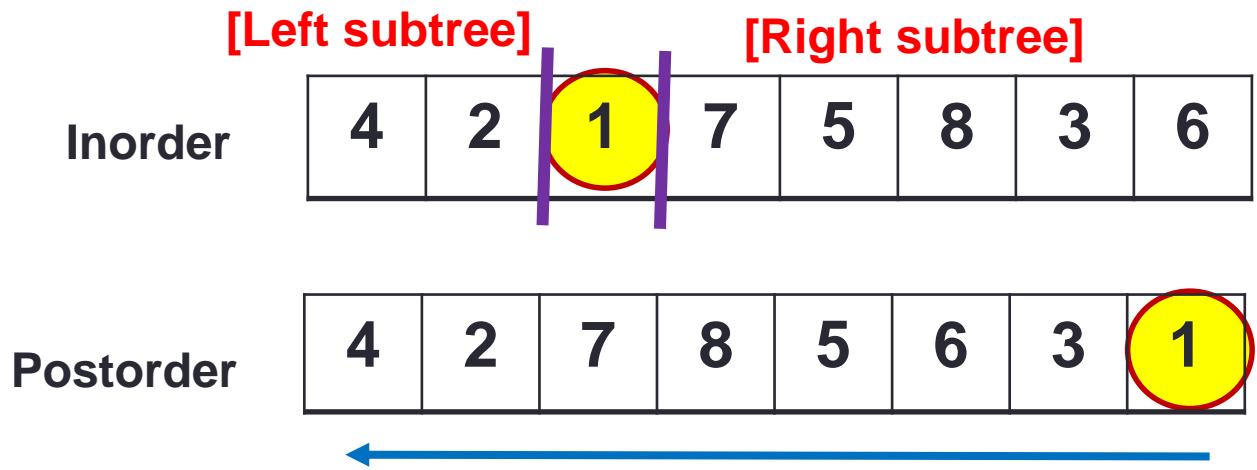
# STEPS

- Inorder traversal is a special traversal that helps us to identify a node and its left and right subtree
- Postorder traversal always gives us the root node as the last element

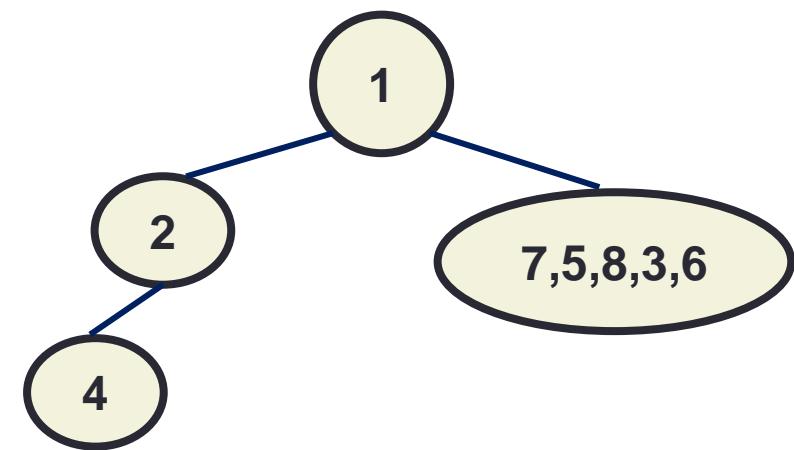
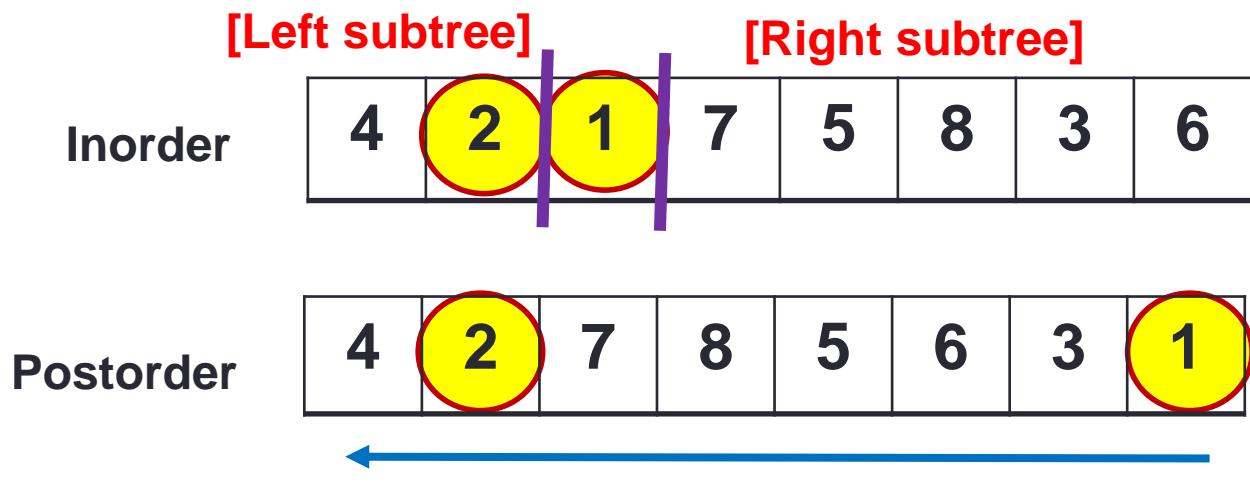
Inorder [Left, Root, Right]	<table border="1"><tr><td>4</td><td>2</td><td>1</td><td>7</td><td>5</td><td>8</td><td>3</td><td>6</td></tr></table>	4	2	1	7	5	8	3	6
4	2	1	7	5	8	3	6		
Postorder [Left, Right, Root]	<table border="1"><tr><td>4</td><td>2</td><td>7</td><td>8</td><td>5</td><td>6</td><td>3</td><td>1</td></tr></table>	4	2	7	8	5	6	3	1
4	2	7	8	5	6	3	1		



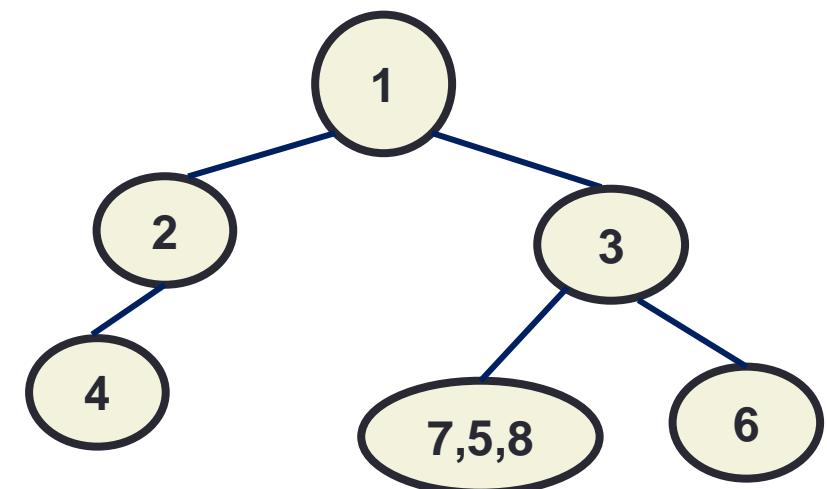
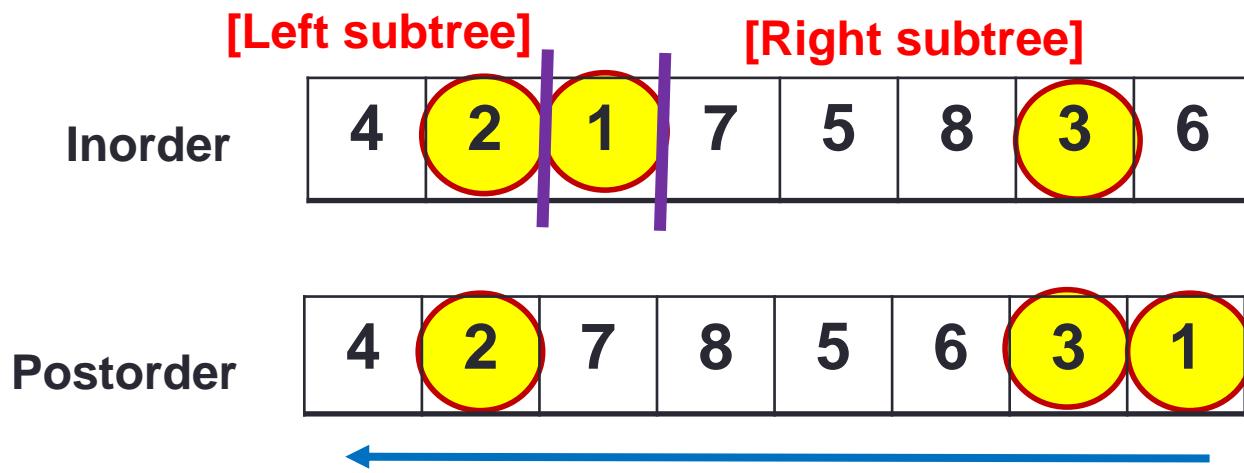
# STEPS



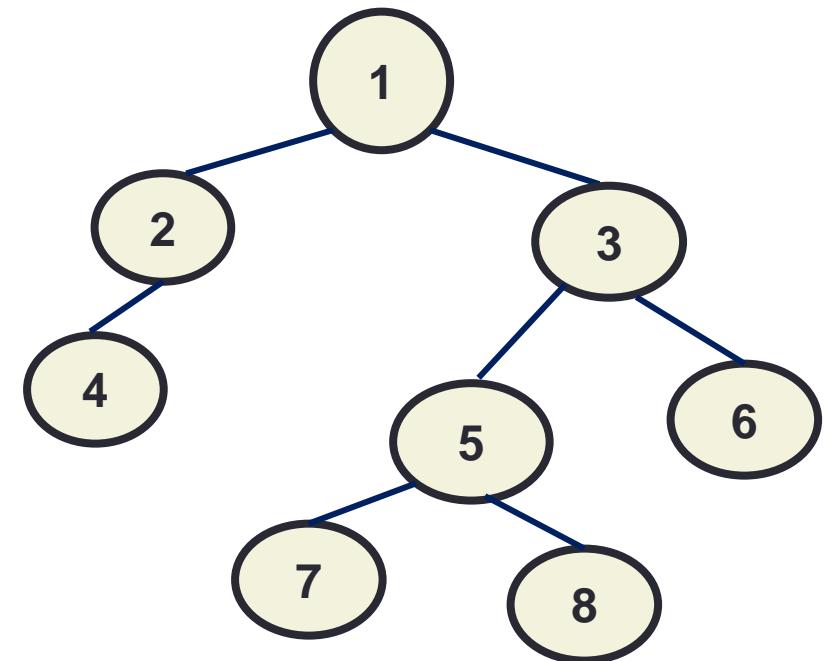
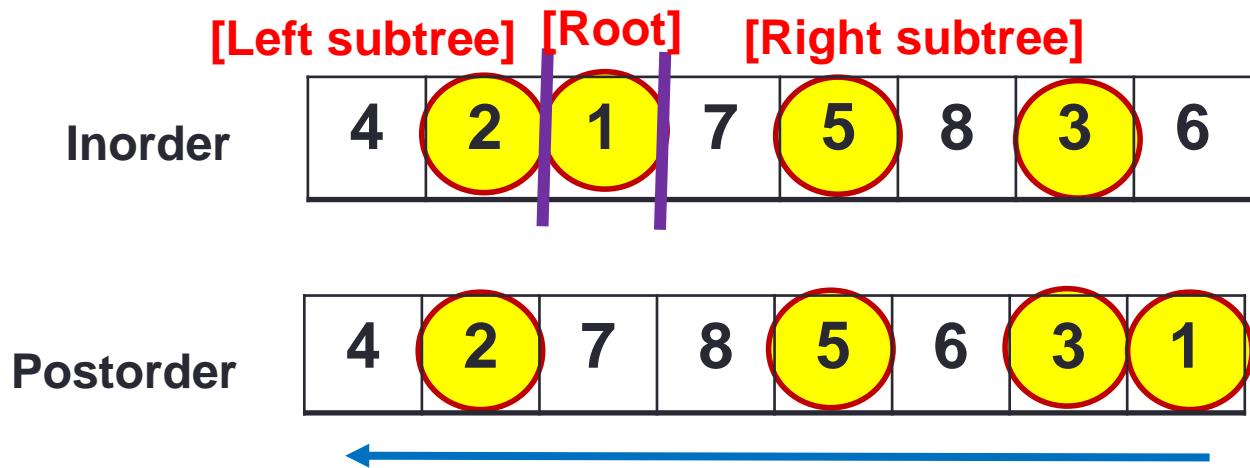
# STEPS



# STEPS



# STEPS



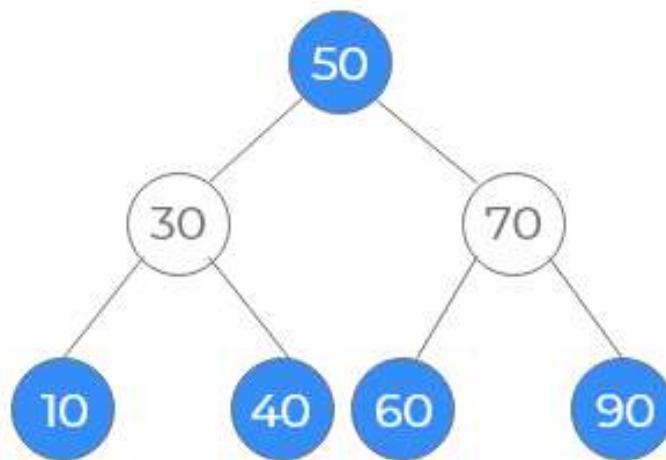
# Try it on your own

Input :-

Inorder - [ 10 30 40 50 60 70 90 ]

Postorder - [ 10 40 30 60 90 70 50 ]

Output :-



# CONSTRUCT BINARY TREE FROM PREORDER AND POSTORDER

---

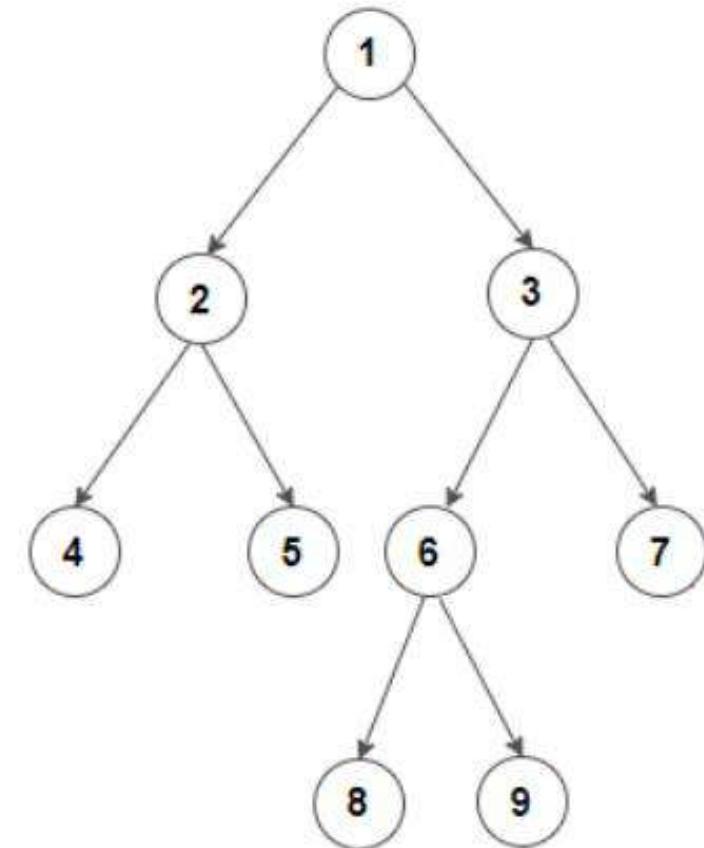
# Given Preorder and Postorder construct Full Binary tree

**Preorder**  
[Root, Left, Right]

1	2	4	5	3	6	8	9	7
---	---	---	---	---	---	---	---	---

**Postorder**  
[Left, Right, Root]

4	5	2	8	9	6	7	3	1
---	---	---	---	---	---	---	---	---



# STEP 1: Find root node

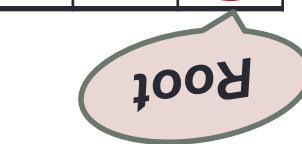
**Preorder**  
[Root, Left, Right]

Root

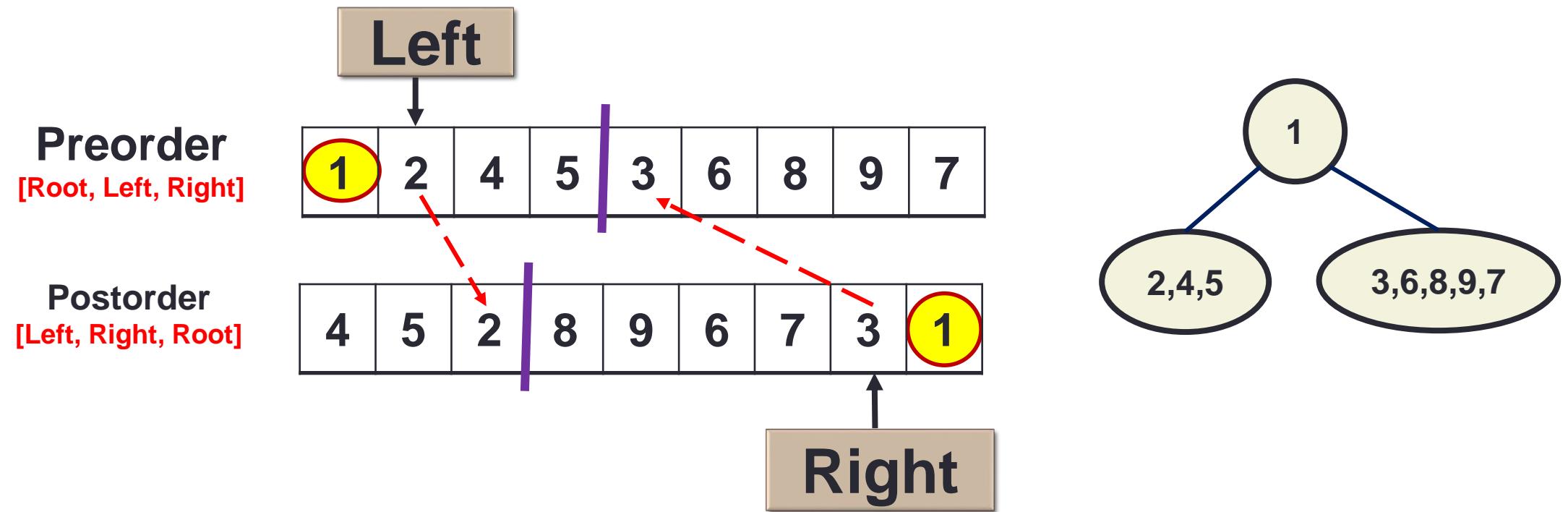
1	2	4	5	3	6	8	9	7
---	---	---	---	---	---	---	---	---

**Postorder**  
[Left, Right, Root]

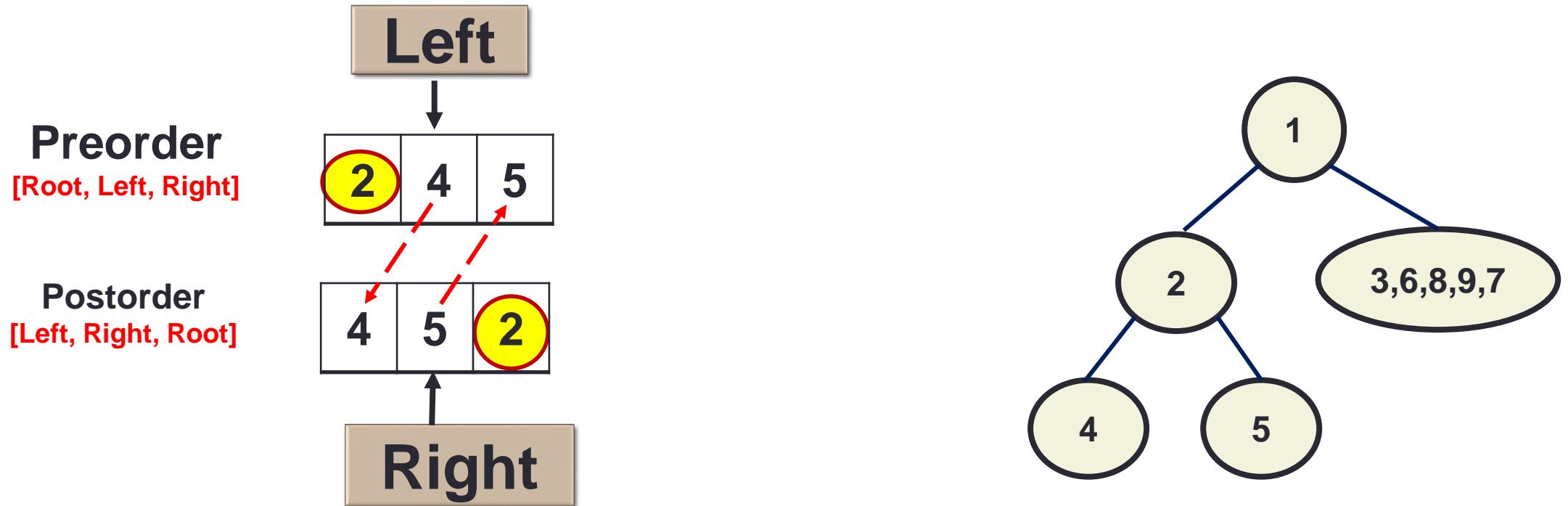
4	5	2	8	9	6	7	3	1
---	---	---	---	---	---	---	---	---



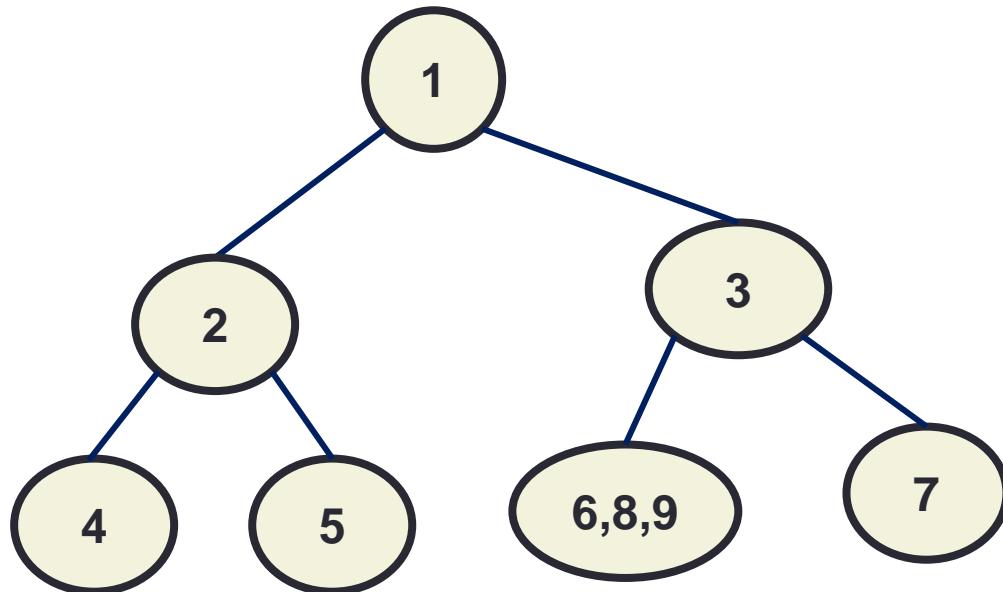
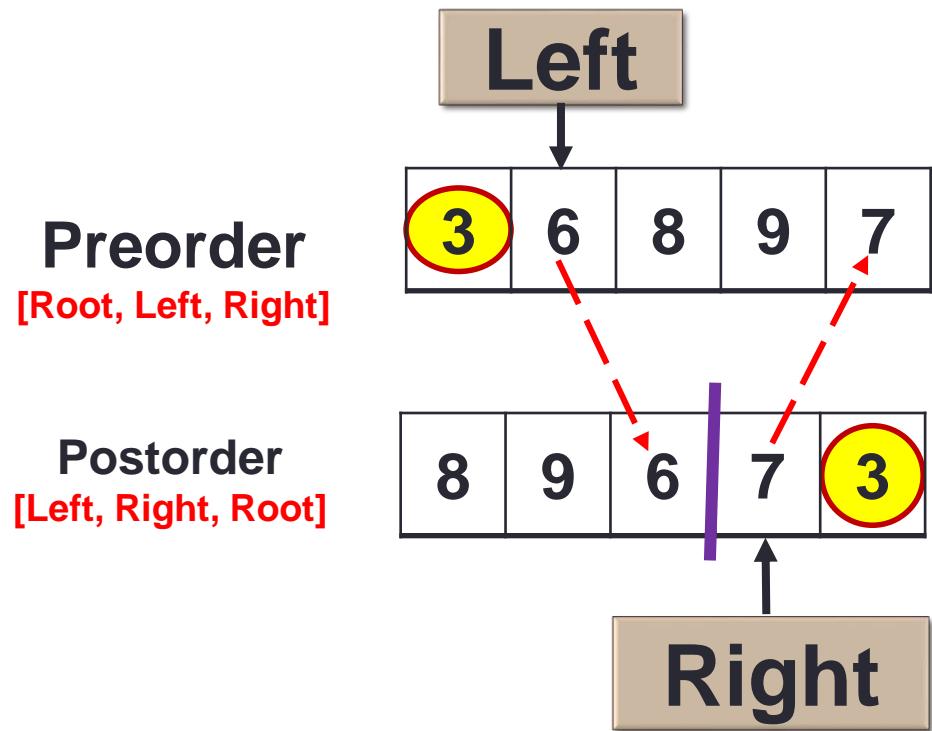
## STEP 2: Find Left and Right subtree of the root node



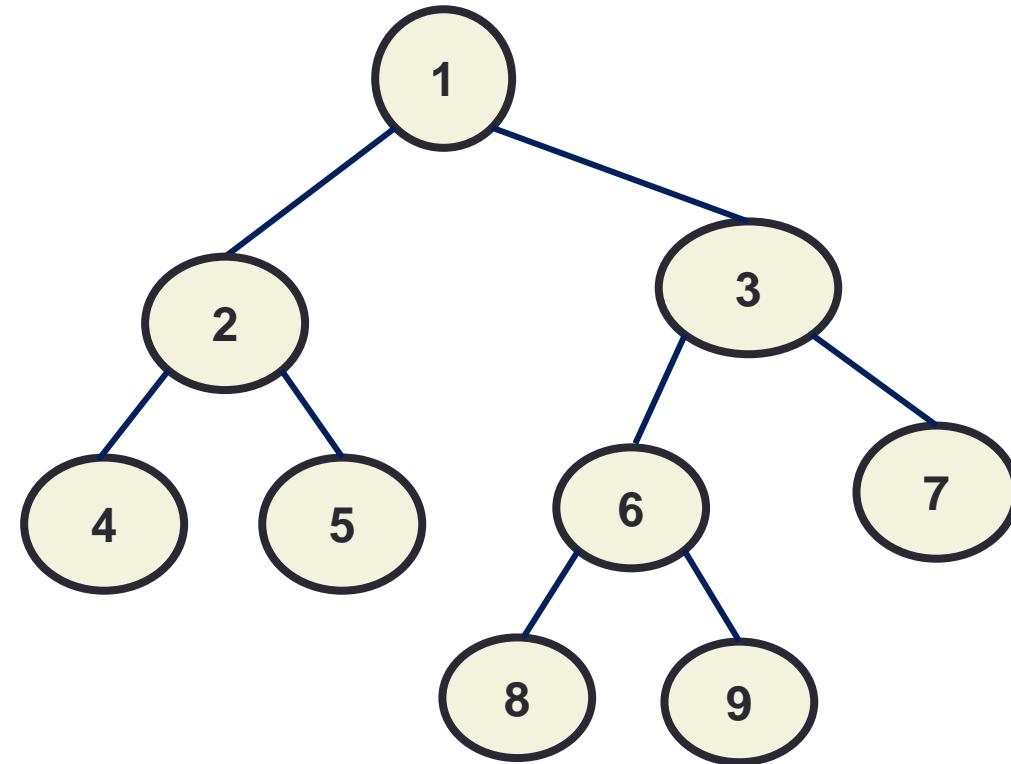
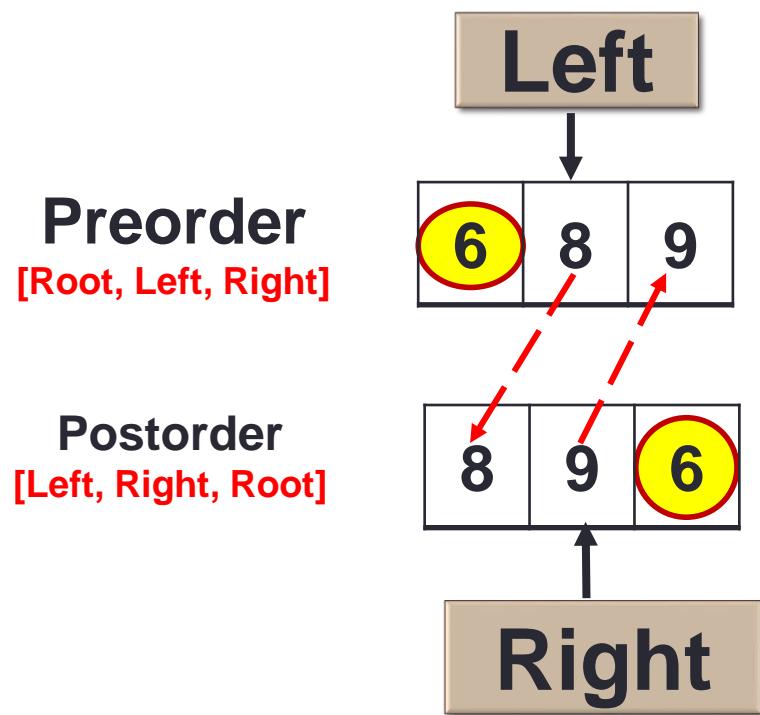
## STEP 3: Repeat the process on left subtree



## STEP 4: Repeat the process on Right subtree



## STEP 4: Repeat the process on Right subtree

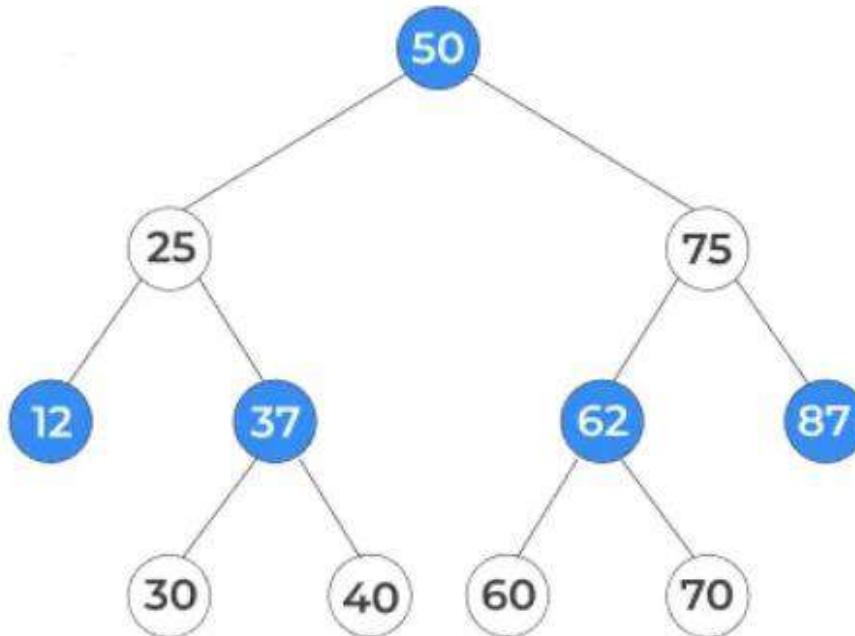


**Final Binary Tree**

Try it on your own

**Preorder:** 50, 25, 12, 37, 30, 40, 75, 62, 60, 70, 87

**Postorder:** 12, 30, 40, 37, 25, 60, 70, 62, 87, 75, 50





# **BCS202L- Data Structures and Algorithms**

**Dr. Priyanka N**  
**Assistant Professor Senior Grade I**  
**School of Computer Science & Engineering**  
**VIT, Vellore.**

# **BCS202L- Data Structures and Algorithms**

- Module-1:Algorithm Analysis
- Module-2: Linear Data Structures
- Module-3: Searching and Sorting
- Module-4: Trees
- Module-5: Graphs
- Module-6: Hashing
- Module-7: Heaps and AVL Trees

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# **BCS202L- Data Structures and Algorithms**

## **Text Books:**

- I.** Mark A. Weiss, Data Structures & Algorithm Analysis in C++, 4 th Edition, 2013,Pearson Education.

## **Reference Books:**

- I.** Alfred V. Aho, Jeffrey D. Ullman and John E. Hopcroft, Data Structures and Algorithms,1983, Pearson Education.
- 2.** Horowitz, Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, 2008, 2<sup>nd</sup> Edition, Universities Press.
- 3.** Thomas H. Cormen, C.E. Leiserson, R L. Rivest and C. Stein, Introduction to Algorithms, 2009, 3<sup>rd</sup> Edition, MIT Press.

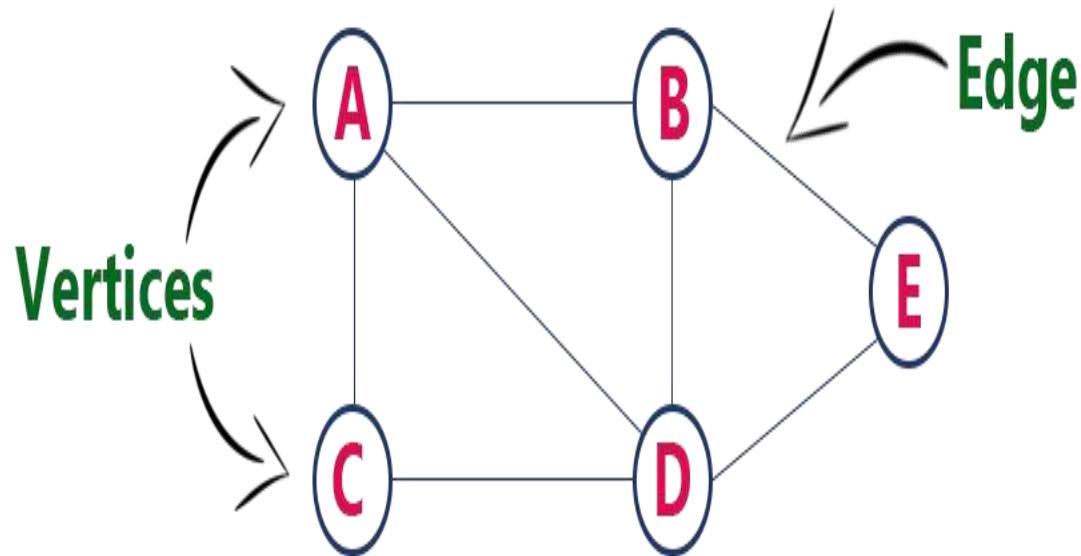
# Module 5: Graphs( 6 Hours)

- Terminology and Representation
- Graph Traversal
  - DFS
  - BFS
- Minimum Spanning Tree
  - Prim's Algorithm
  - Kruskal's Algorithm
- Single Source Shortest Path
  - Dijkstra's Algorithm

# Graph- Definition

- Graph is a non-linear data structure.
- It contains a set of points known as **nodes (or vertices)** and a set of links known as **edges (or Arcs)**. Here edges are used to connect the vertices.
- A graph is defined as “**G**” consist of two sets **V** and **E** where **V** is a finite set of non empty set of vertices and **E** is set of pairs of vertices. These pairs are called edges.
- Generally, a graph **G** is represented as **G = (V,E)**, where **V is set of vertices** and **E is set of edges**.

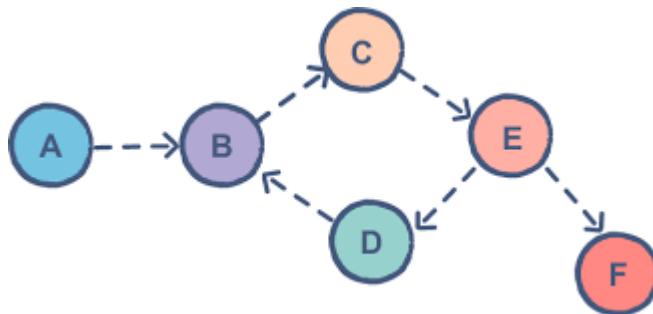
# Graph- Example – 5 vertices and 7 Edges



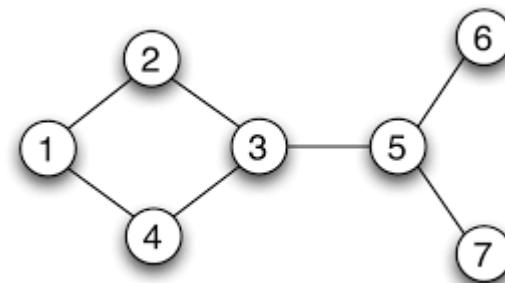
# Graph Terminology

- **Types of Graph**

- Directed: A graph with only directed edges is said to be directed graph.
- Undirected Graph: A graph with only undirected edges is said to be undirected graph.



Directed Graph

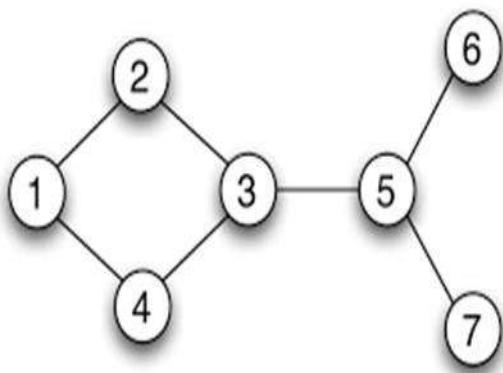


Un Directed Graph

# Graph Terminology

- **Vertex**

- Individual data element of a graph is called as **Vertex**. **Vertex** is also known as **node**. In the below example graph,



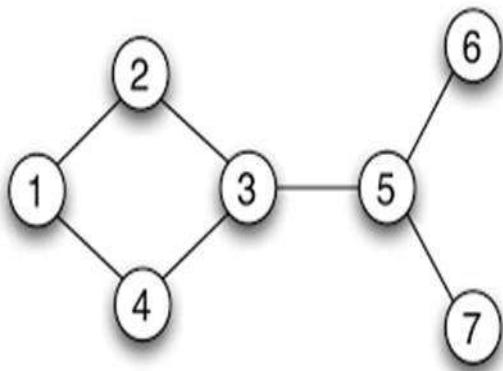
**For Example:**

1,2,3,4,5,6,7 are known as vertices.

# Graph Terminology

- **Edges**

- An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex).



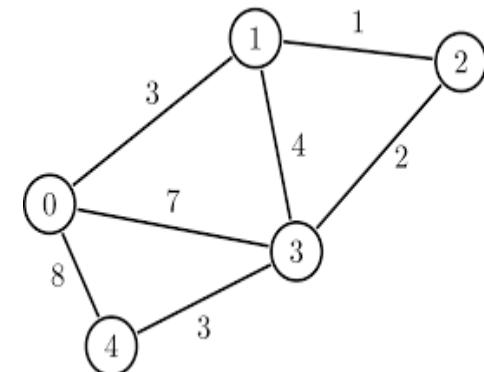
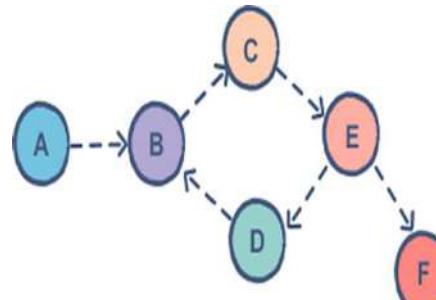
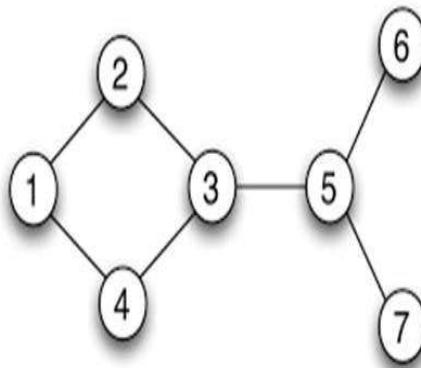
**For Example:**

The edges in the graphs are  
(1,2),(1,4),(2,3),(3,4),(3,5),(5,6), (5,7)

# Graph Terminology

## • Types of Edges

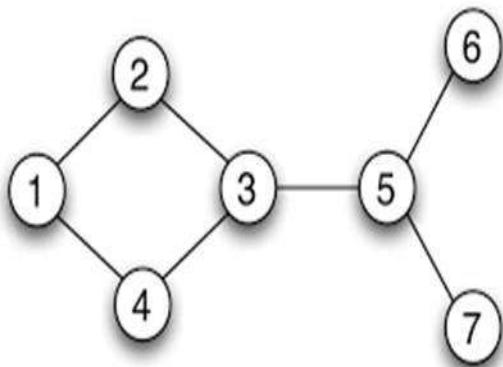
- **Undirected Edge** - An undirected edge is a bidirectional edge. If there is undirected edge between vertices 1 and 2 then edge (1 , 2) is equal to edge (2 , 1).
- **Directed Edge** - A directed edge is a unidirectional edge. If there is directed edge between vertices 1 and 2 then edge (1 , 2) is not equal to edge (2 , 1).
- **Weighted Edge** - A weighted edge is a edge with value (cost) on it.



# Graph Terminology

- **Adjacent**

- If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

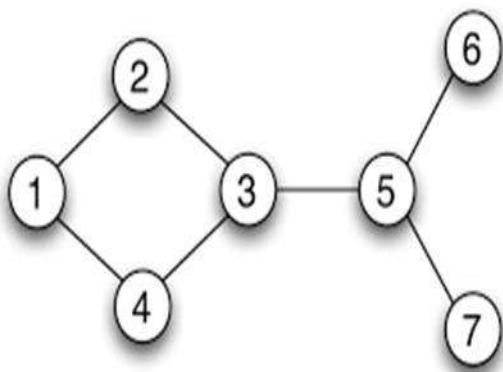


For Example:  
Adjacent Nodes  
for 3 is (2,4,5)

# Graph Terminology

- **Incident**

- Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

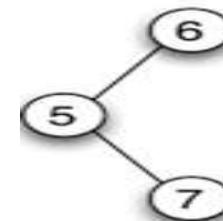
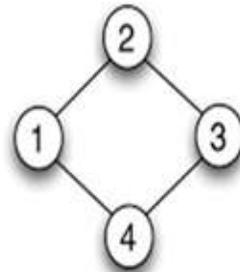
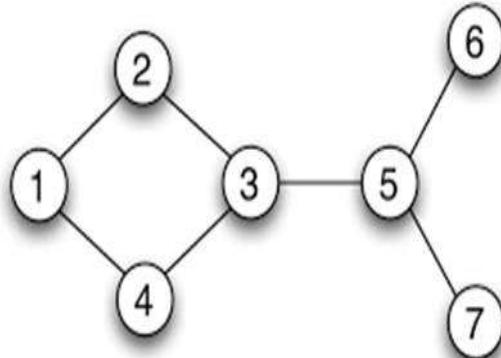


For Example: The edges incident on Vertex 2 is (1,2) and (2,3)

# Graph Terminology

- **Subgraph**

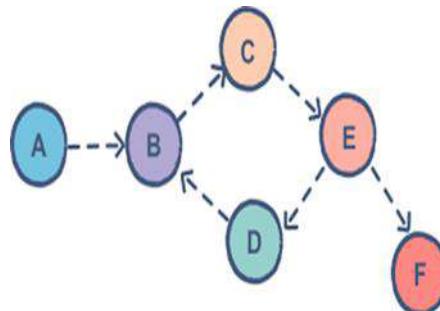
- A Sub graph of  $G$  is  $G'$  such that vertices of  $G'$  belongs to  $G$  and all edges of  $G'$  belongs to  $G$ .



# Graph Terminology

## • Path

- A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.
- Length of the path: Number of edges on the path.



Example Path:

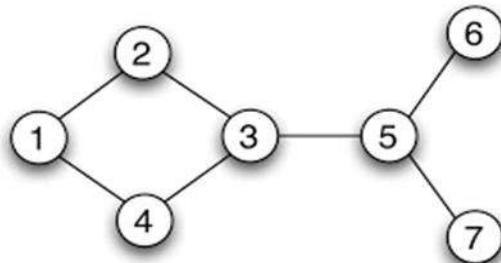
A-B-C-E-F

- Cycle: It is a simple path in which first and last vertices are same.

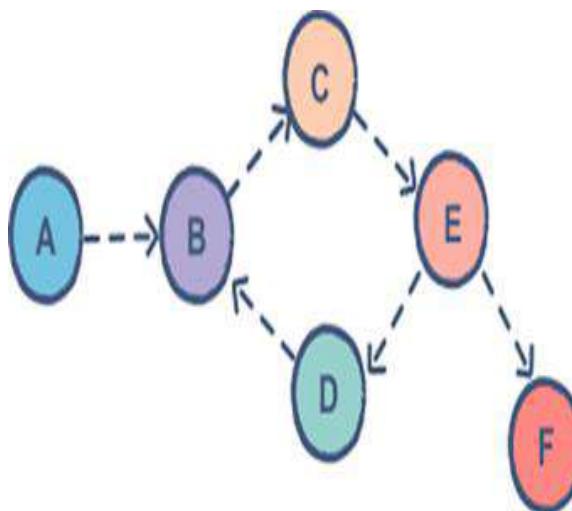
# Graph Terminology

## • Degree of the Vertex

- Total number of edges connected to a vertex is said to be degree of that vertex.



Example: Vertex of 5 is 3



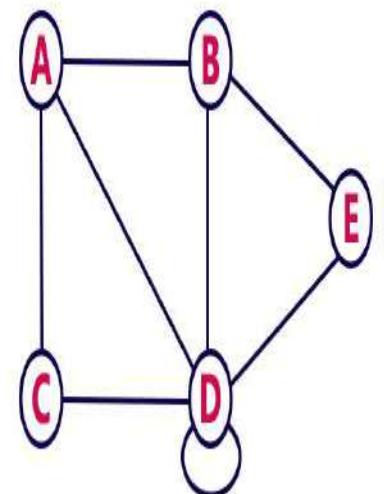
**InDegree :** Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

**OutDegree :** Total number of Outgoing edges connected to a vertex is said to be indegree of that vertex.

# Representation of Graphs

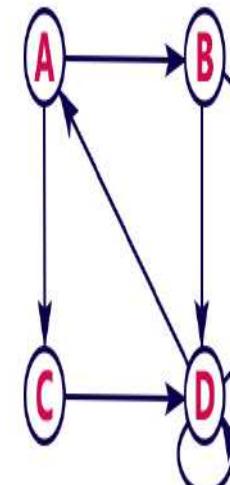
- **Adjacency Matrix**

## Undirected Graph



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

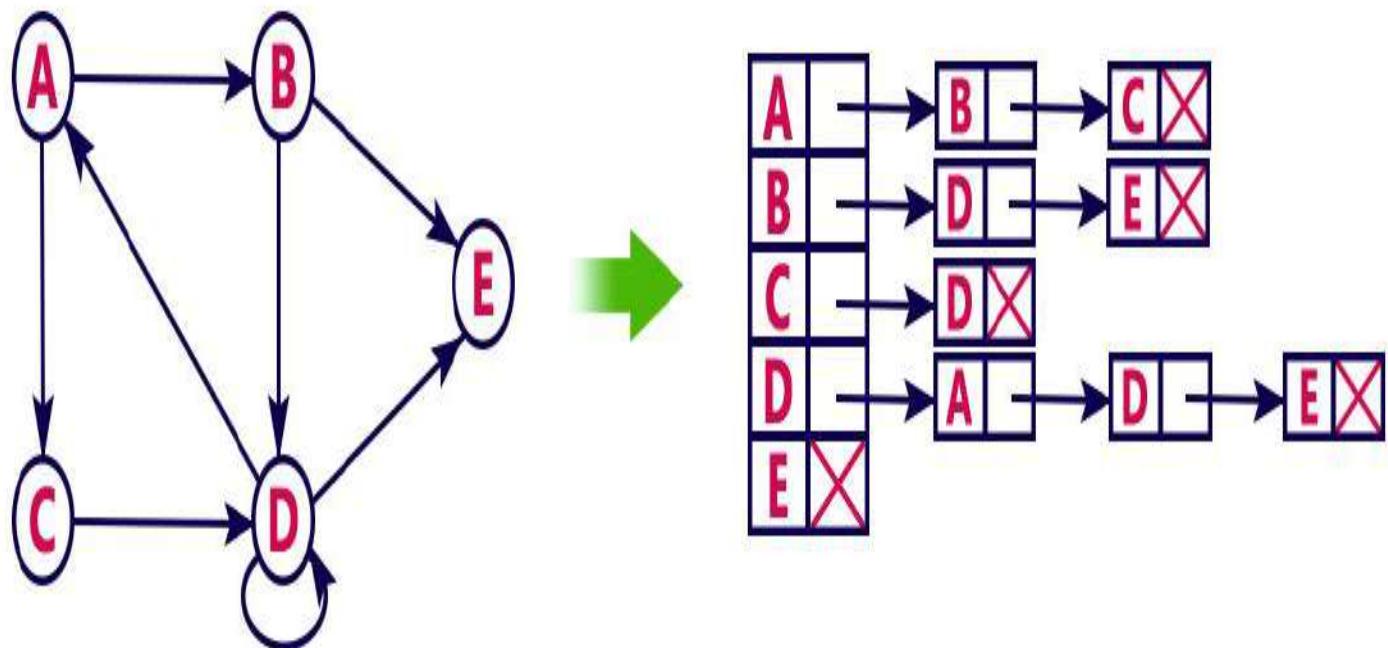
## Directed Graph



	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

# Representation of Graphs

- **Adjacency List**



# **BCS202L- Data Structures and Algorithms**

**Dr. Priyanka N**

**Assistant Professor Senior Grade 1**

**School of Computer Science & Engineering**

**VIT, Vellore.**

# BCS202L- Data Structures and Algorithms

- Module-1: Algorithm Analysis
- Module-2: Linear Data Structures
- Module-3: Searching and Sorting
- Module-4: Trees
- Module-5: Graphs
- Module-6: Hashing
- Module-7: Heaps and AVL Trees

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# BCS202L- Data Structures and Algorithms

## Text Books:

1. Mark A. Weiss, Data Structures & Algorithm Analysis in C ++ , 4 th Edition, 2013, Pearson Education.

## Reference Books:

1. Alfred V. Aho, Jeffrey D. Ullman and John E. Hopcroft, Data Structures and Algorithms, 1983, Pearson Education.
2. Horowitz, Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, 2008, 2<sup>nd</sup> Edition, Universities Press.
3. Thomas H. Cormen, C.E. Leiserson, R L. Rivest and C. Stein, Introduction to Algorithms, 2009, 3<sup>rd</sup> Edition, MIT Press.

# Module 5: Graphs( 6 Hours)

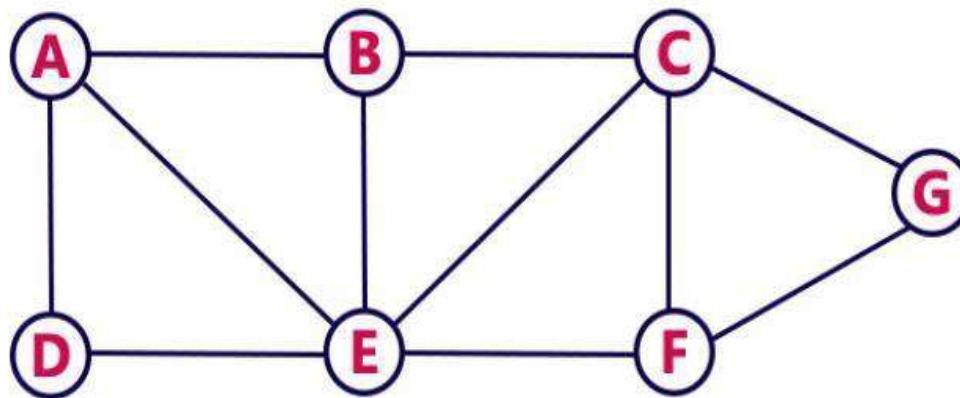
- Terminology and Representation
- **Graph Traversal**
  - DFS
  - BFS
- Minimum Spanning Tree
  - Prim's Algorithm
  - Kruskal's Algorithm
- Single Source Shortest Path
  - Dijkstra's Algorithm

# Graph Traversal

- Graph traversal is a technique used for a searching vertex in a graph.
- Used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.
  - **Depth First Search**
  - **Breadth First Search**

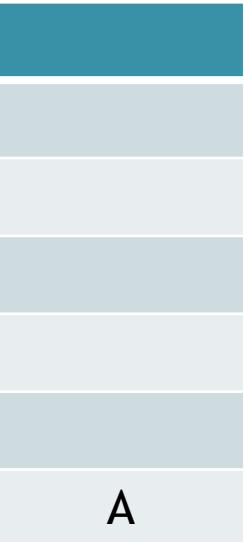
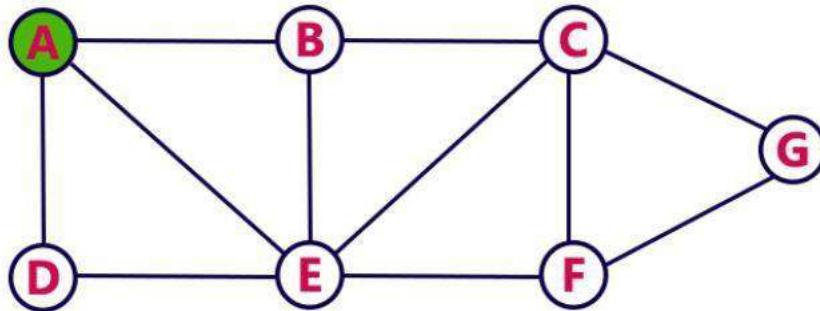
# Depth First Search

- DFS traversal of a graph produces a **spanning tree as a result**
  - **Spanning Tree** is a graph without loops
- 
- Consider the Following Example



# Depth First Search- STEP 1

- Choose the Starting Vertex as A
- VISIT A
- Push A into the Stack

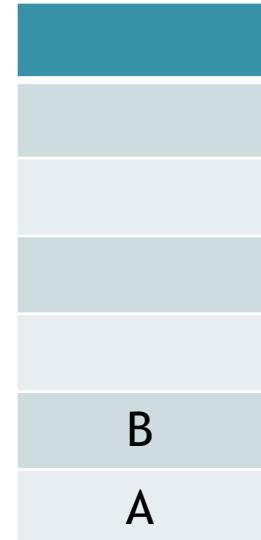
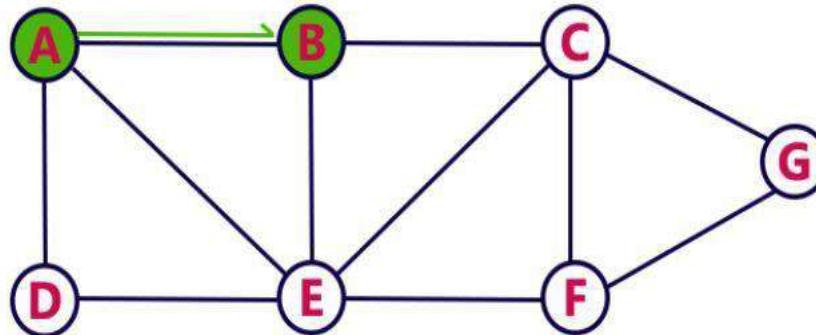


**RESULT:** A

**STACK**

# Depth First Search- STEP 2

- Visit any adjacent vertex of A which is not visited ( taken B)
- VISIT B
- Push newly visited B into the Stack

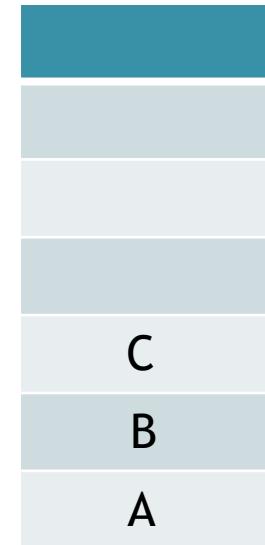
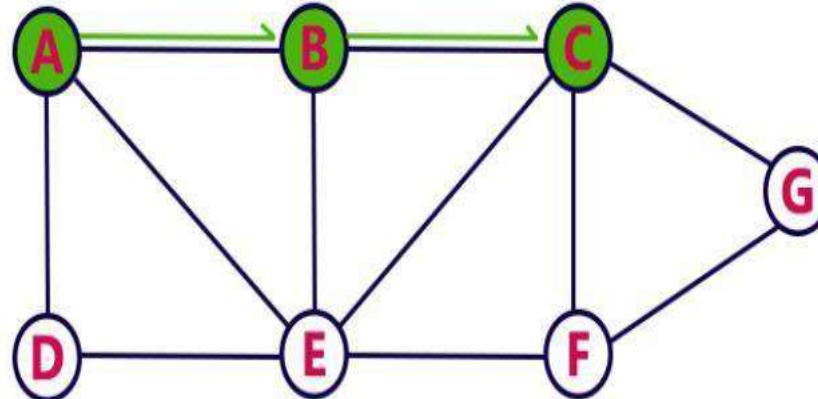


**RESULT:** A B

**STACK**

# Depth First Search- STEP 3

- Visit any adjacent vertex of B which is not visited ( taken C)
- VISIT C
- Push newly visited C into the Stack

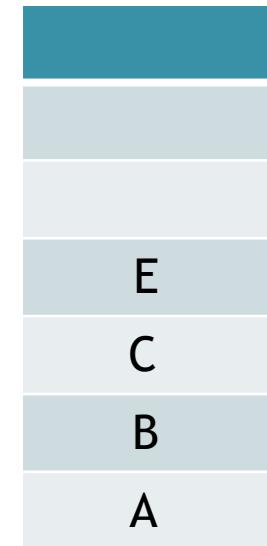
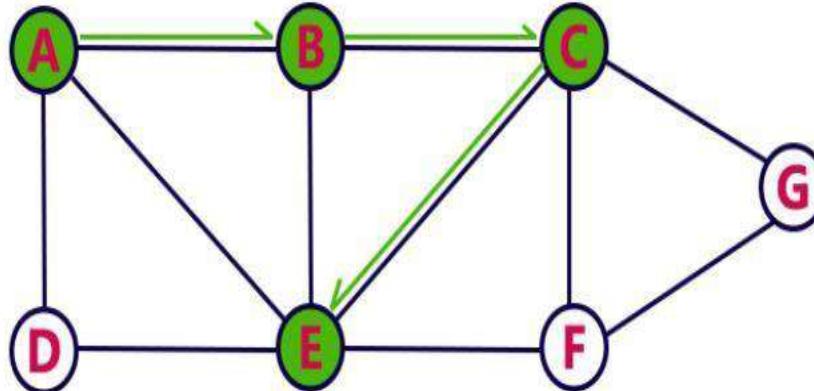


**RESULT:** A B C

**STACK**

# Depth First Search- STEP 4

- Visit any adjacent vertex of C which is not visited ( taken E)
- VISIT E
- Push newly visited E into the Stack

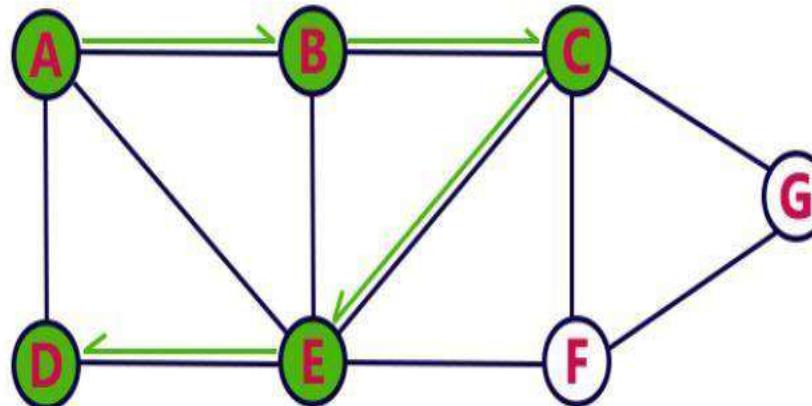


**RESULT:** A B C E

**STACK**

# Depth First Search- STEP 5

- Visit any adjacent vertex of E which is not visited ( taken D)
- VISIT D
- Push newly visited D into the Stack

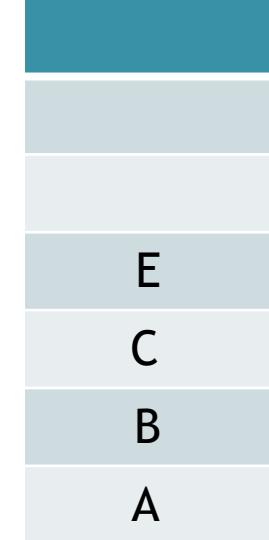
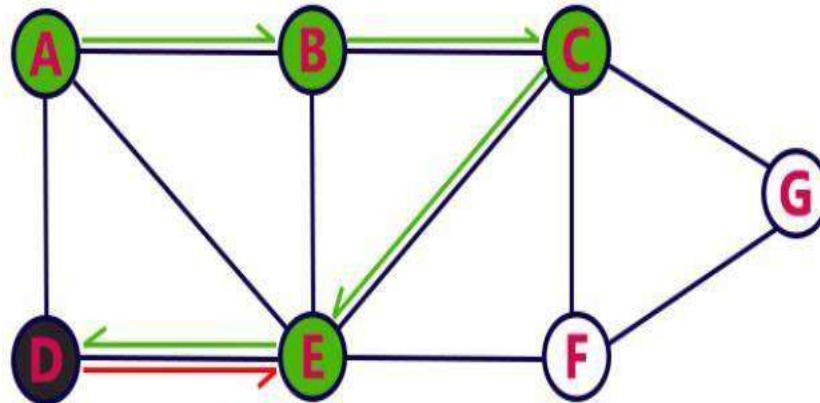


**RESULT:** A B C E D

**STACK**

# Depth First Search- STEP 6

- No new vertex to be visited from D
- POP D

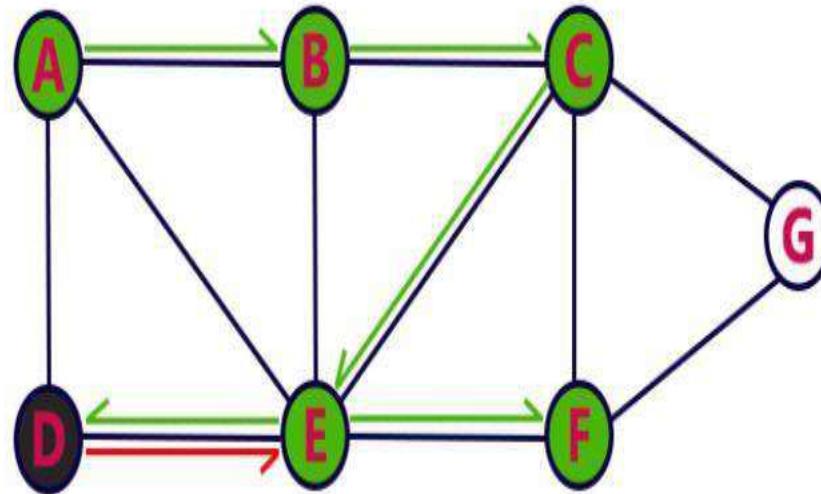


**RESULT:** A B C E D

**STACK**

# Depth First Search- STEP 7

- Visit any adjacent vertex of E which is not visited ( taken F)
- VISIT F
- Push newly visited F into the Stack

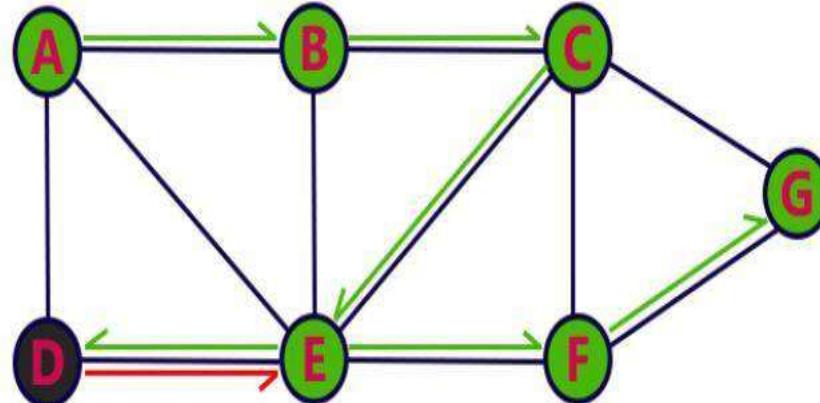


**RESULT:** A B C E D F

**STACK**

# Depth First Search- STEP 8

- Visit any adjacent vertex of F which is not visited ( taken G)
- VISIT G
- Push newly visited G into the Stack

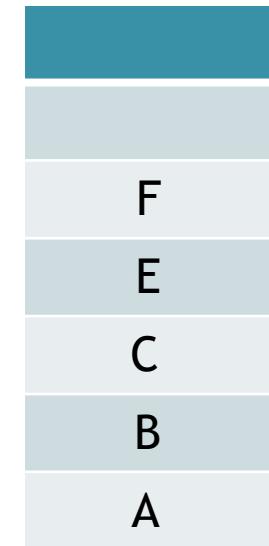
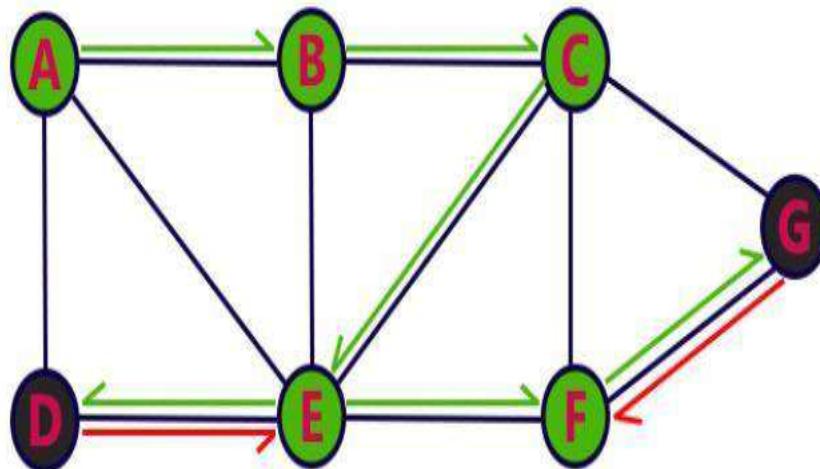


**RESULT:** A B C E D F G

**STACK**

# Depth First Search- STEP 9

- No new vertex to be visited from G
- POP G

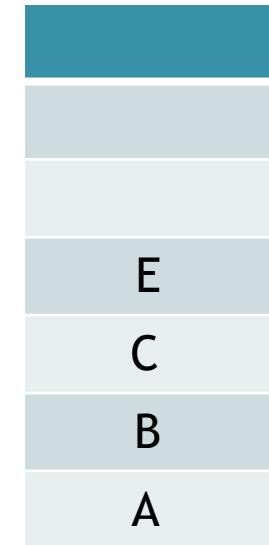
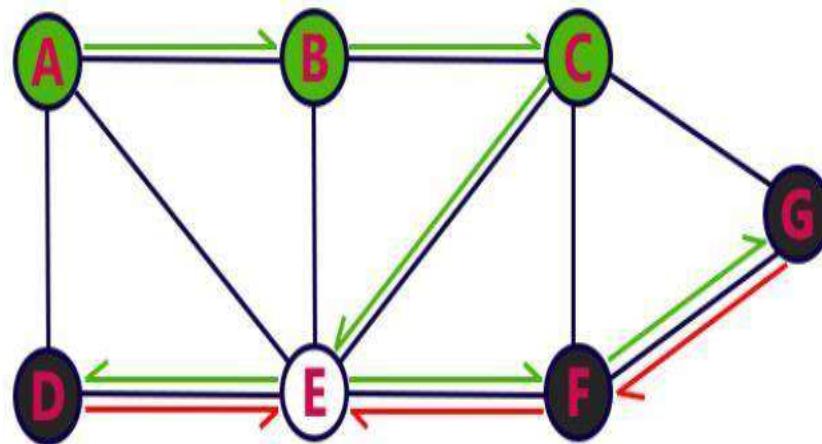


**RESULT:** A B C E D F G

**STACK**

# Depth First Search- STEP 10

- No new vertex to be visited from F
- POP F

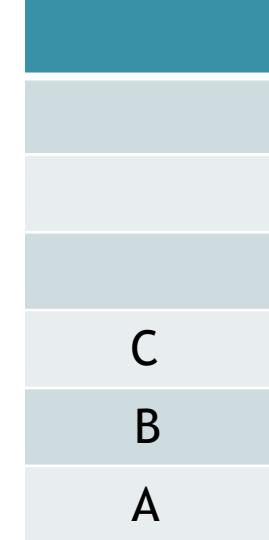
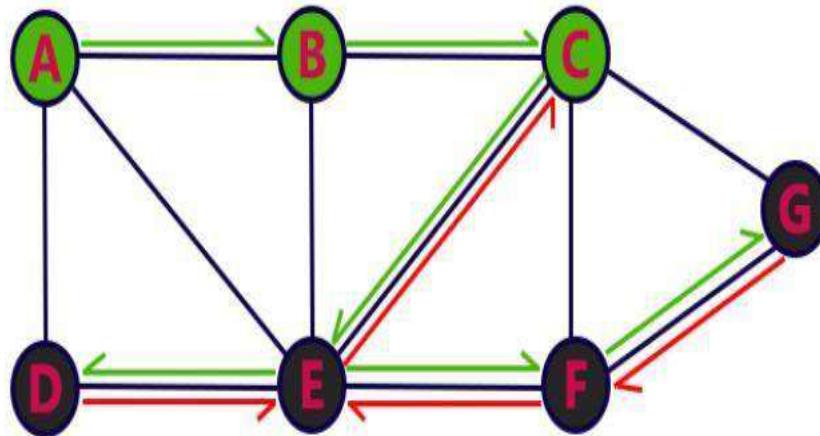


**RESULT:** A B C E D F G

**STACK**

# Depth First Search- STEP 11

- No new vertex to be visited from E
- POP E

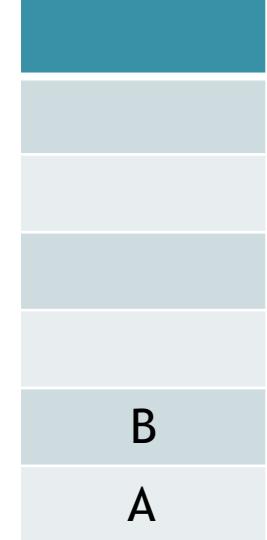
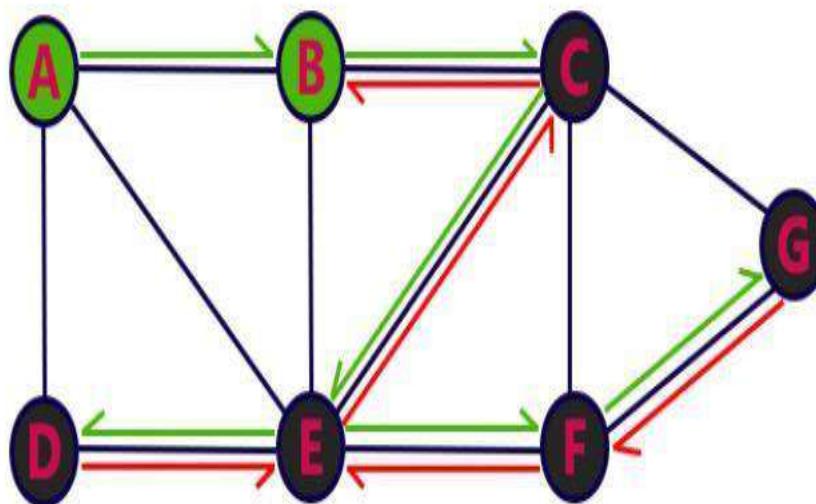


**RESULT:** A B C E D F G

**STACK**

# Depth First Search- STEP 12

- No new vertex to be visited from C
- POP C

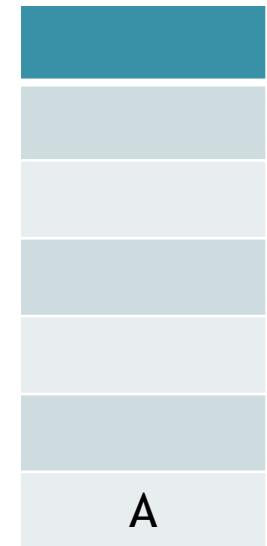
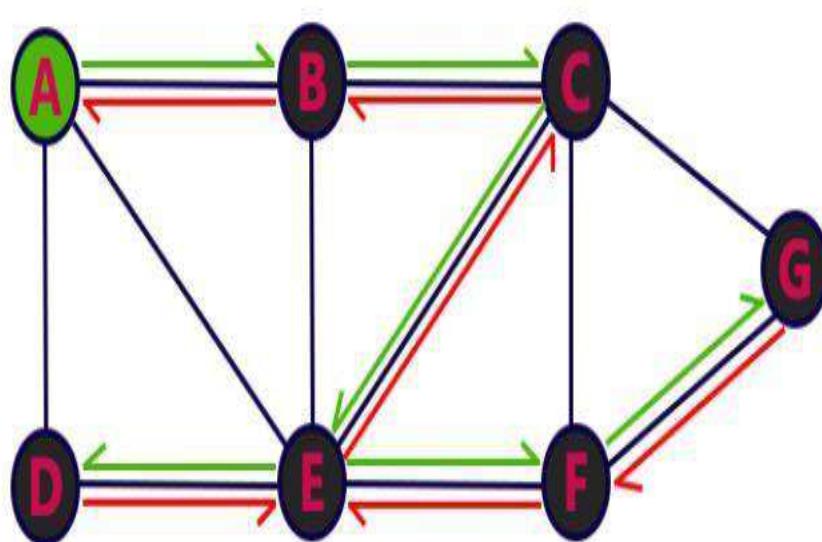


**RESULT:** A B C E D F G

**STACK**

# Depth First Search- STEP 13

- No new vertex to be visited from B
- POP B

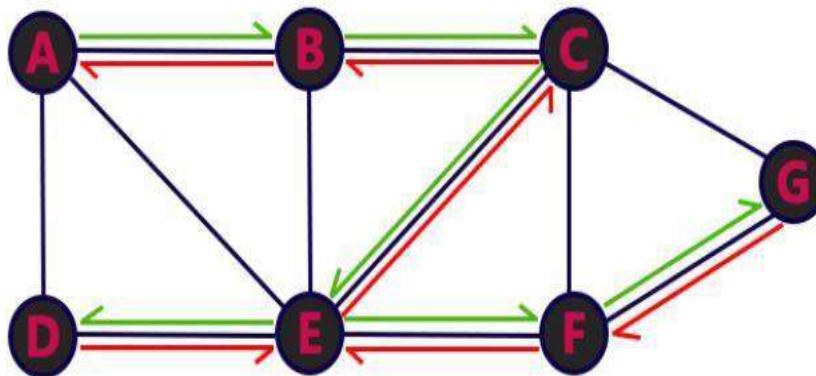


**RESULT:** A B C E D F G

**STACK**

# Depth First Search- STEP 14

- No new vertex to be visited from A
- POP A

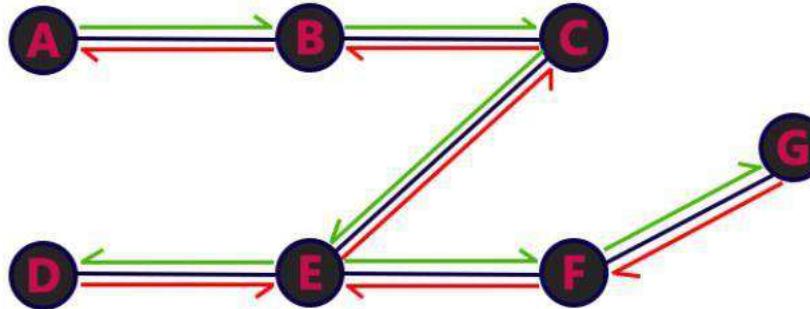


**RESULT:** A B C E D F G

Stack Empty, End the DFS traversal

STACK

# DFS Traversal - Result



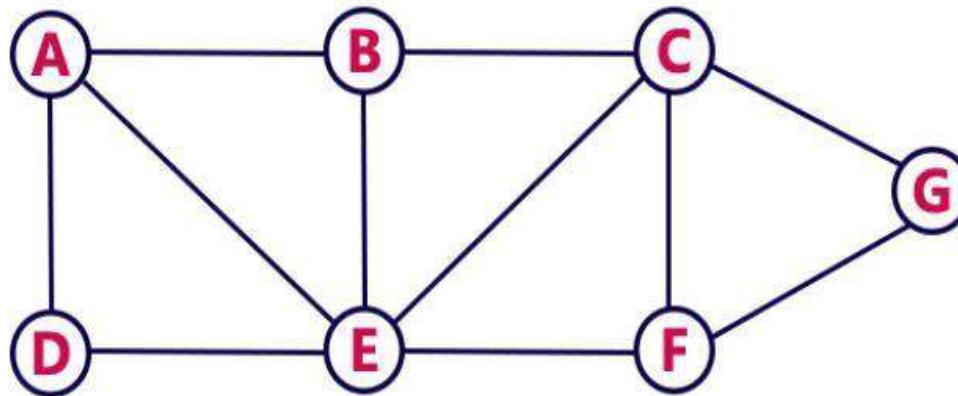
$A \rightarrow B \rightarrow C \rightarrow E \rightarrow D \rightarrow F \rightarrow G$

## Algorithm DFS(V) // n- number of vertices

```
{  
int adj;  
vis[v]=1; // Array to track the visited node  
printf("%d", v);  
for(adj=1;adj<=n;adj++) // Iterating adjacent vertices  
of "v"  
{  
    if(vis[adj]==0) // Adjacent node is not visited  
    {  
        if(adj[v][adj]==1)  
        {  
            dfs(adj);  
        }  
    }  
}
```

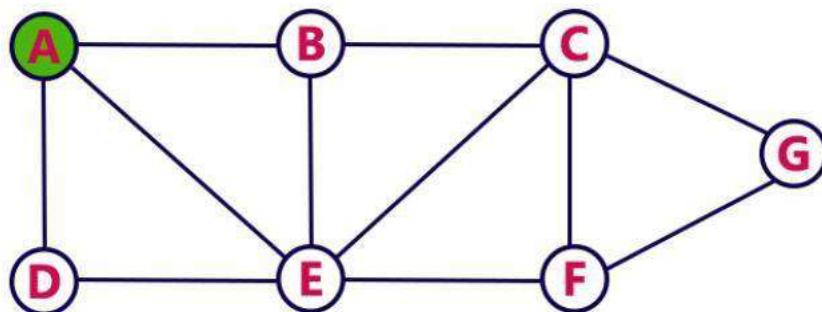
# Breadth First Search

- BFS traversal of a graph produces a **spanning tree** as result. **Spanning Tree** is a graph without loops.
- Consider the Following Example



# Breadth First Search- STEP1

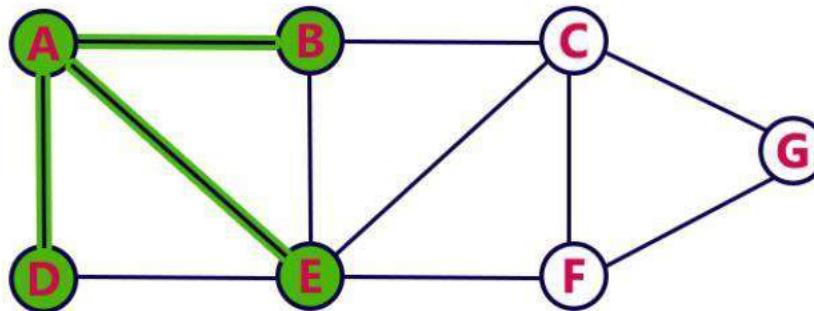
- Choose the Starting Vertex as A
- VISIT A
- Insert A into the Queue



QUEUE: A |   |   |   |

# Breadth First Search- STEP2

- Visit all adjacent vertices of A which are not visited(Here:D,E,B)
- Insert newly visited vertices into the queue and deleteA from the Queue

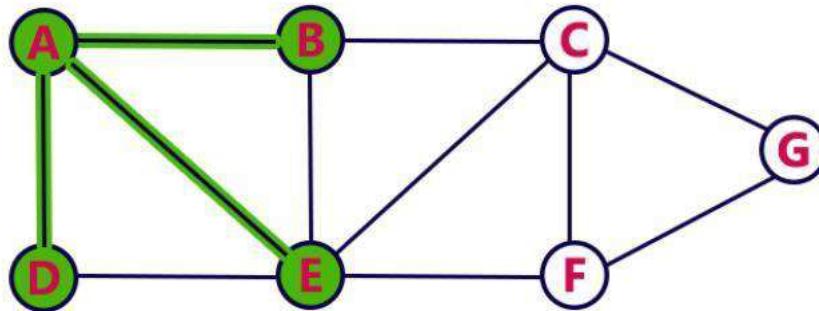


QUEUE: [ ] | D | E | B | [ ] | [ ]

RESULT: A

# Breadth First Search- STEP 3

- Visit all adjacent vertices of D which are not visited(No Vertex)
- Delete D from the Queue

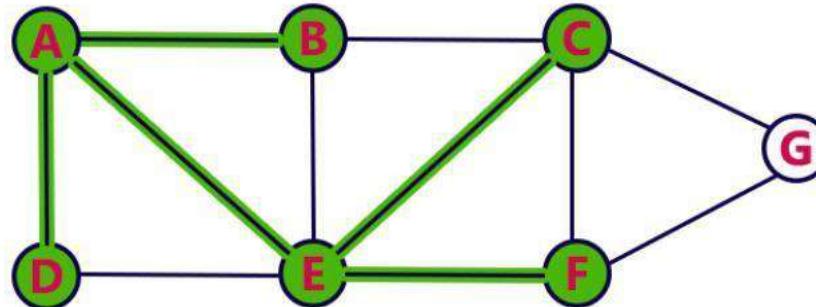


QUEUE: [ ] | [ ] | E | B | [ ] | [ ]

**RESULT:** A D

# Breadth First Search- STEP4

- Visit all adjacent vertices of E which are not visited(Here C, F)
- Insert newly visited vertices into the queue and delete E from the Queue

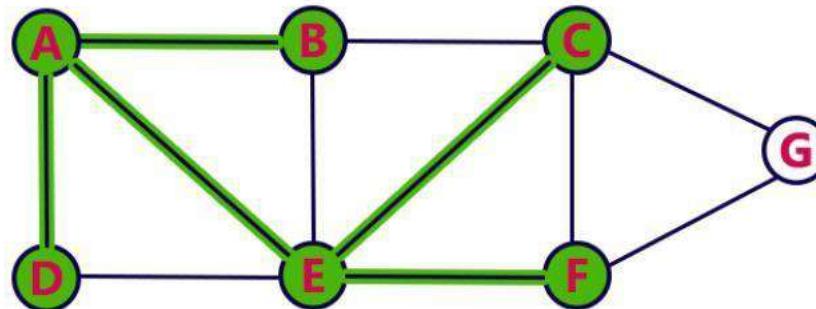


QUEUE: [ ] [ ] [ ] B [ ] C [ ] F [ ]

RESULT: A D E

# Breadth First Search- STEP5

- Visit all adjacent vertices of B which are not visited(No Vertex)
- Delete B from the Queue

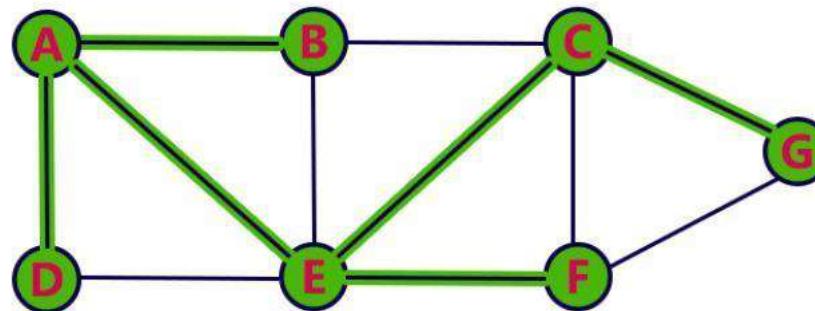


QUEUE: [ ] [ ] [ ] [ ] C F [ ]

RESULT: A D E B

# Breadth First Search- STEP6

- Visit all adjacent vertices of C which are not visited(Here G)
- Insert newly visited vertices into the queue and delete C from the Queue



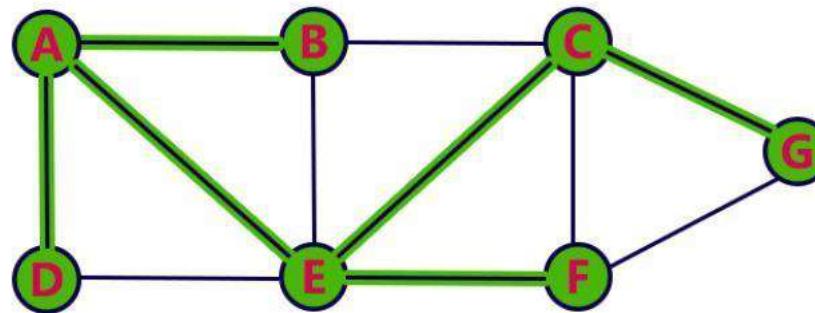
QUEUE:



RESULT: A D E B C

# Breadth First Search- STEP7

- Visit all adjacent vertices of F which are not visited(Here No Vertex)
- Delete F from the Queue

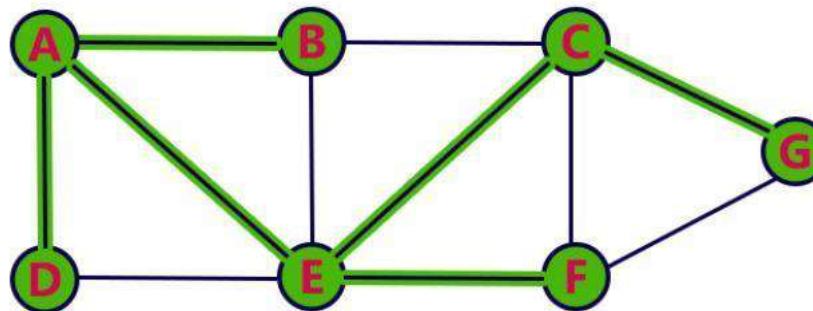


QUEUE: [ ] [ ] [ ] [ ] [ ] [ ] G

RESULT: A D E B C F

# Breadth First Search- STEP8

- Visit all adjacent vertices of G which are not visited(Here No Vertex)
- Delete G from the Queue



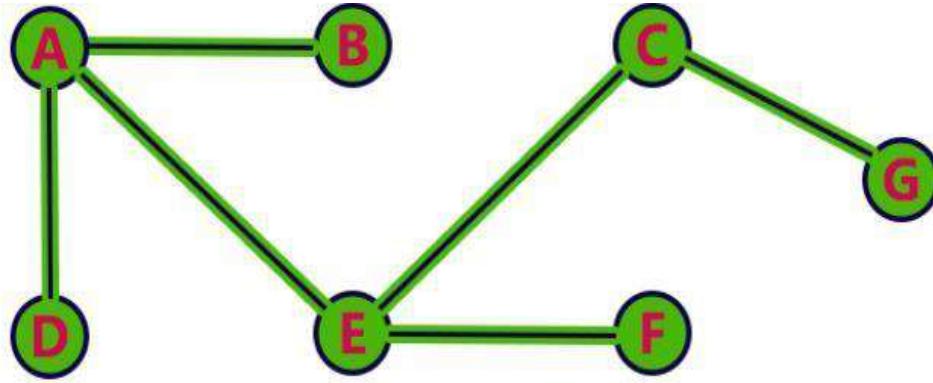
QUEUE:



RESULT: A D E B C F G

Queue Empty, End the Process

# BFS Traversal - Result



A → D → E → B → C → F → G

Breadth First Search(v)

{

U=v; // Maintaining a queue of adjacent vertices for “v”

visited[v]=1; // Array to track the visited node

while U is not empty

{

v<-dequeue from U

for each vertices ‘W’ adjacent from U do

{

if(visited[W]=0) then

{

Add ‘W’ to the Queue; // W is also unexplored vertices

visited[W]=1;

}

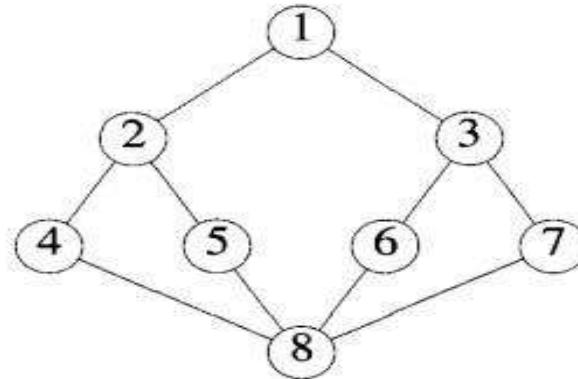
} // end of for loop

}

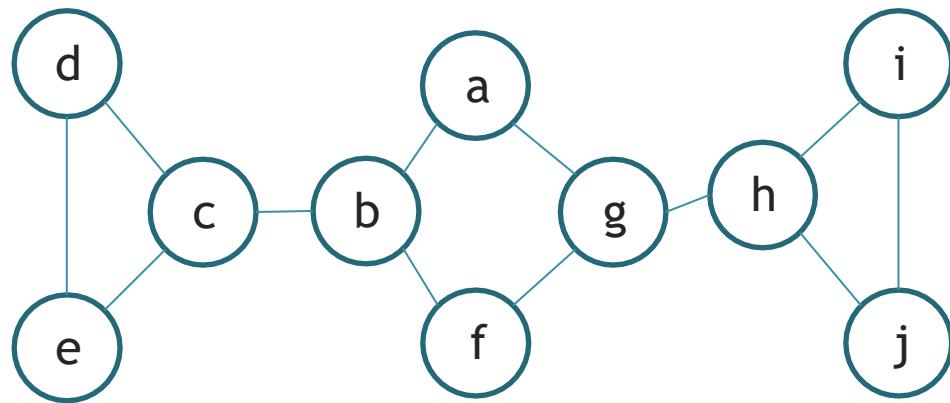
}

# Sample Exercises

- Problem 1



- Problem 2



# Key Differences Between BFS and DFS

- 1.BFS is a vertex-based algorithm while DFS is an edge-based algorithm.
- 2.The queue data structure is used in BFS. On the other hand, DFS uses stack or recursion.
- 3.Memory space is efficiently utilized in DFS while space utilization in BFS is not effective.
- 4.BFS is the optimal algorithm while DFS is not optimal.
- 5.DFS constructs narrow and long trees. As against, BFS constructs wide and short trees.



# BCS202L- Data Structures and Algorithms

Dr. Priyanka N  
Assistant Professor Senior Grade 1  
School of Computer Science & Engineering  
VIT, Vellore.

# BCS202L- Data Structures and Algorithms

- Module-1: Algorithm Analysis
- Module-2: Linear Data Structures
- Module-3: Searching and Sorting
- Module-4: Trees
- Module-5: Graphs
- Module-6: Hashing
- Module-7: Heaps and AVL Trees

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the k <sup>th</sup> minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# BCS202L- Data Structures and Algorithms

## Text Books:

1. Mark A. Weiss, Data Structures & Algorithm Analysis in C ++ ,4 th Edition, 2013,Pearson Education.

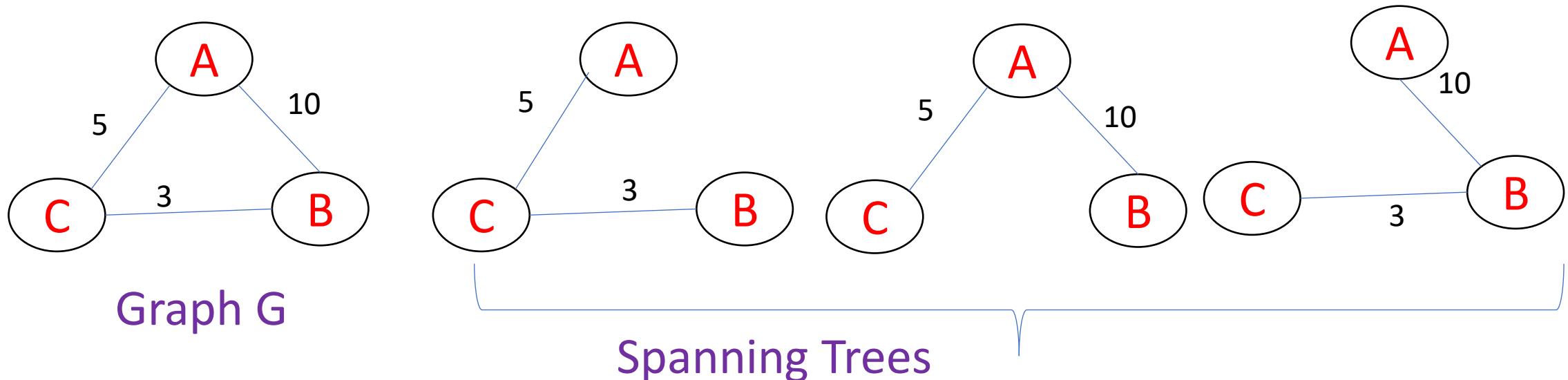
## Reference Books:

1. Alfred V. Aho, Jeffrey D. Ullman and John E. Hopcroft, Data Structures and Algorithms,1983, Pearson Education.
2. Horowitz, Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, 2008,2<sup>nd</sup> Edition, Universities Press.
3. Thomas H. Cormen, C.E. Leiserson, R L. Rivest and C. Stein, Introduction to Algorithms,2009,3<sup>rd</sup> Edition,MIT Press.

# Introduction to Spanning Trees

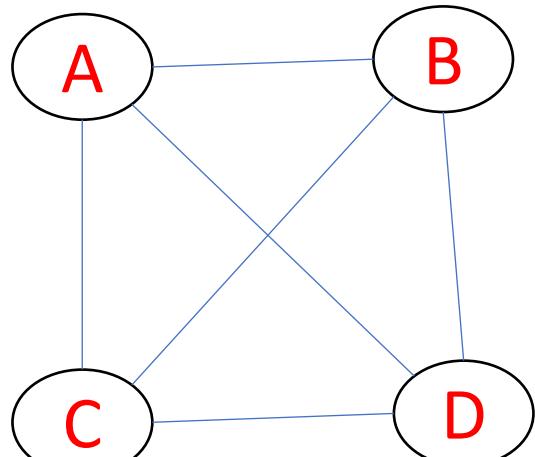
- A spanning tree is a subset of graph G, which has all the vertices covered with minimum possible number of edges.
- A spanning does not have cycles as well as every vertex is connected.
- Hence, every connected undirected graph has at least one spanning tree.

No. of spanning trees for a complete undirected graphs  $n^{(n-2)}$   
where n is number of vertices.

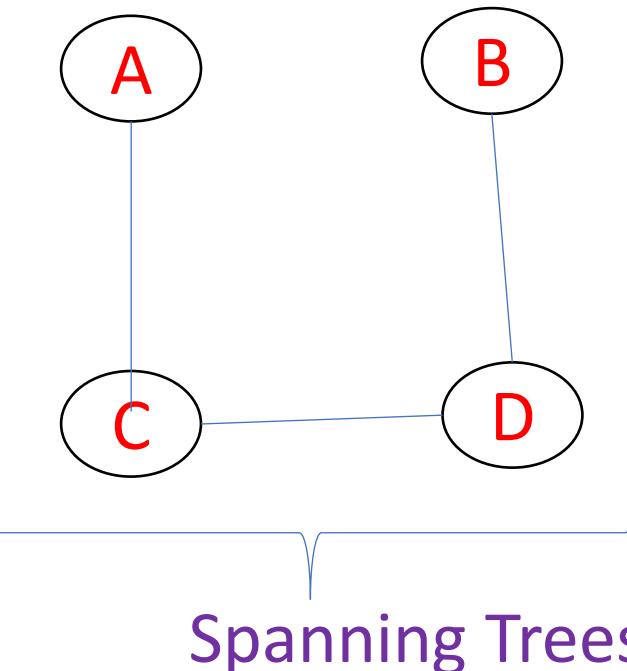


# Examples

No. of edges in a spanning tree are  $n-1$ , where  $n$  is number of vertices.

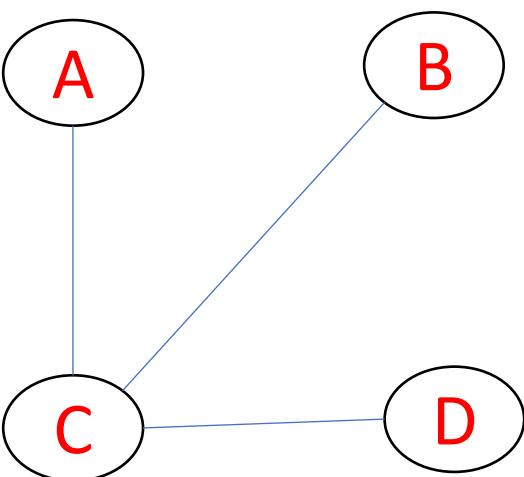
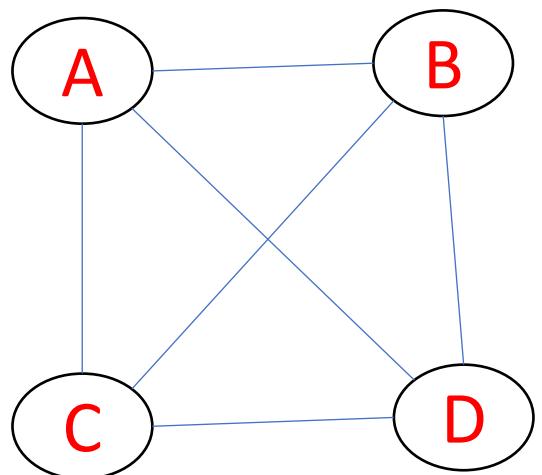


Graph G



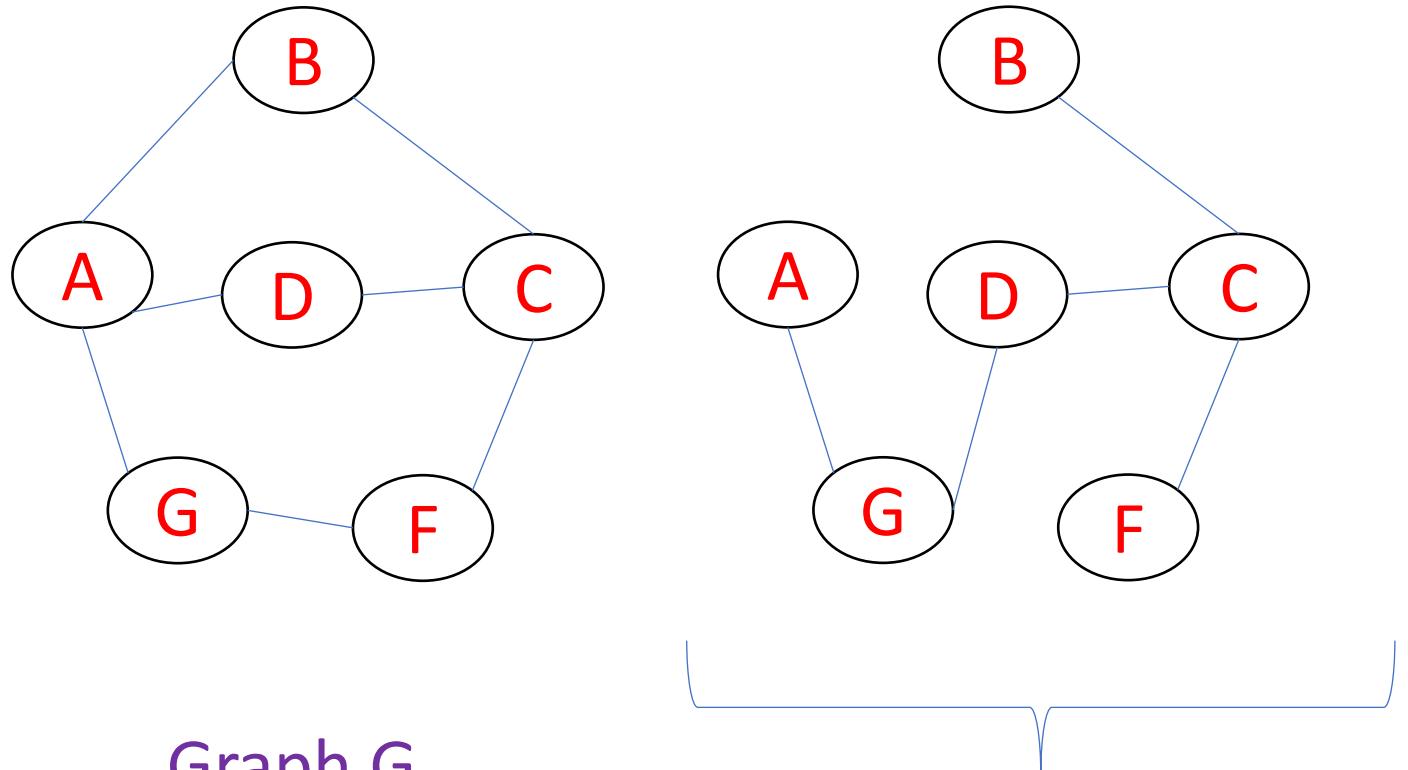
Spanning Trees

## Examples



# Examples

No. of edges in a spanning tree are  $n-1$ , where  $n$  is number of vertices.



Graph G

Spanning Trees

# Properties of spanning trees

- A connected graph G can have more than one spanning tree.
- A disconnected graph do not have spanning trees.
- All possible spanning trees of graph G have the same number of  $(n-1)$  edges and vertices.
- A spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, it means the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit, it means the spanning tree is maximally acyclic.

## Mathematical Properties of spanning trees

- Spanning tree has  $n-1$  edges, where n is the number of nodes (vertices).
- A complete graph can have maximum  $n^{n-2}$  number of spanning trees.

# Applications of spanning trees

- Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common applications of spanning trees are civil network planning, computer network routing protocol, cluster analysis etc.

## Minimum spanning trees

- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

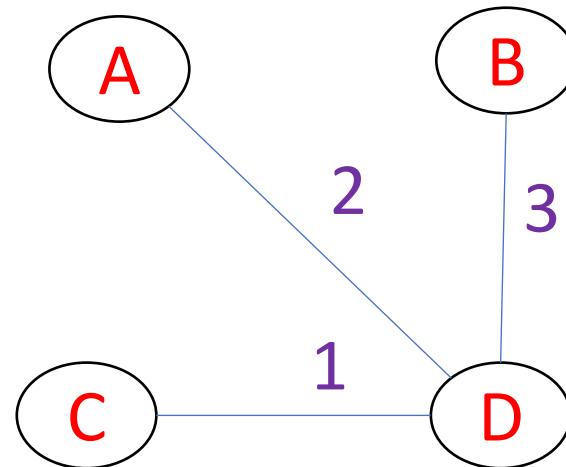
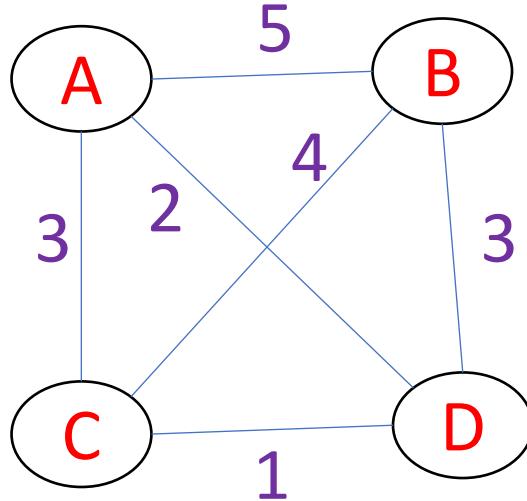
## Minimum spanning tree Algorithm

- Two algorithms for spanning trees are krushkal's Algorithm and Prim's Algorithm.

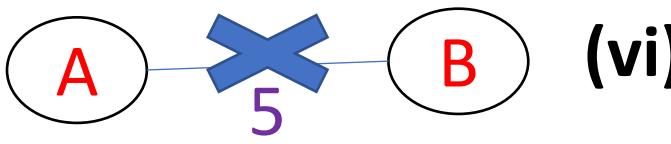
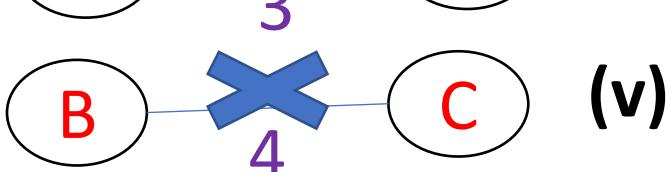
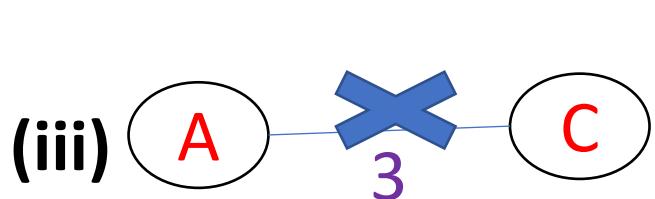
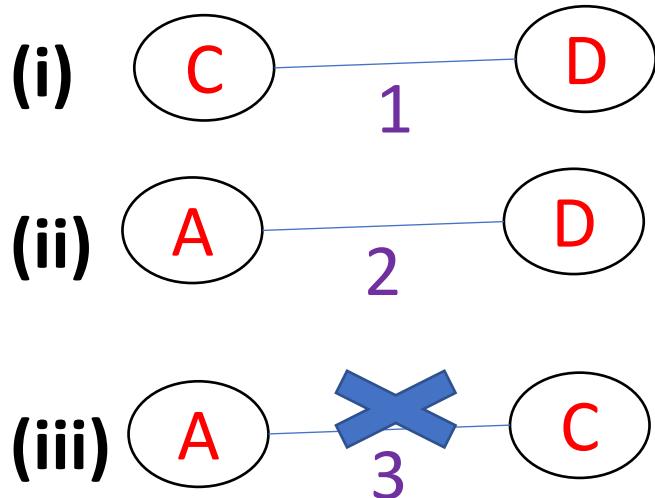
# Minimum Cost Spanning Trees-Krushkal's Algorithm

- 1. Sort all the edges in increasing order of their weight.**
- 2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.**
- 3. Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.**

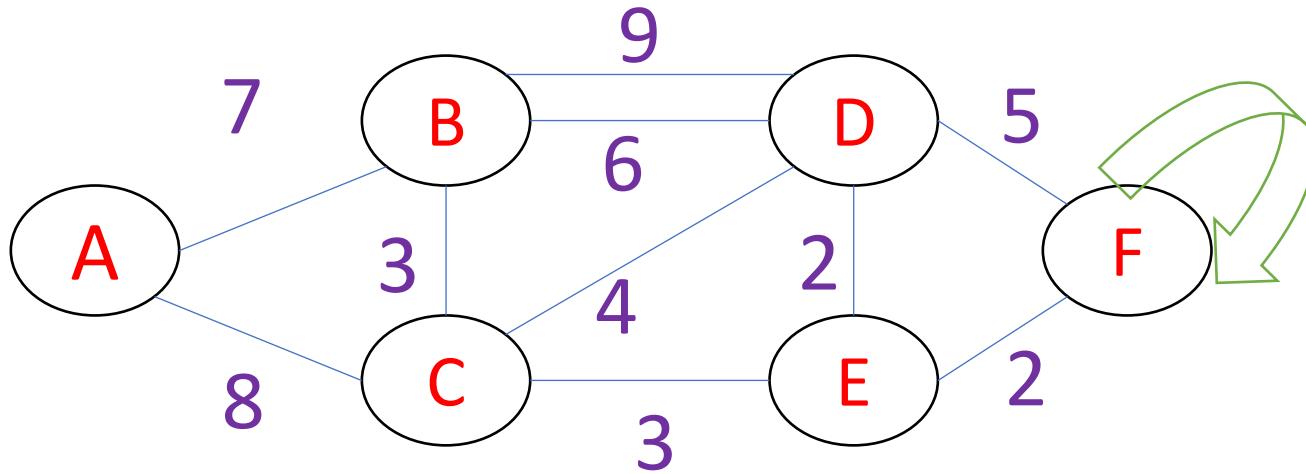
# Minimum Spanning Trees-Krushkal's Algorithm



MST: 6



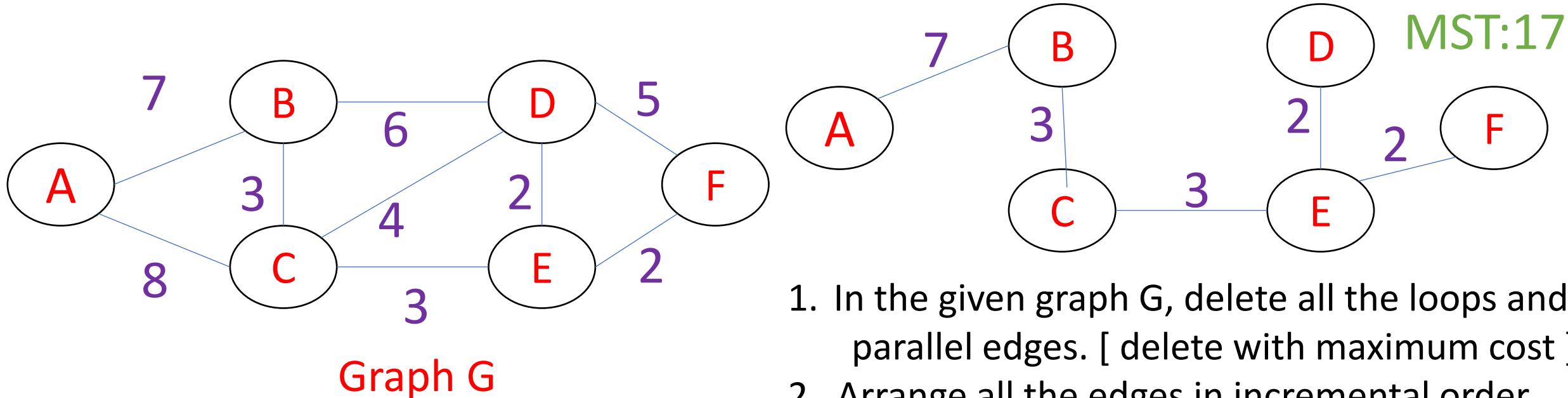
# Minimum Spanning Tree Krushkal's Algorithm Example



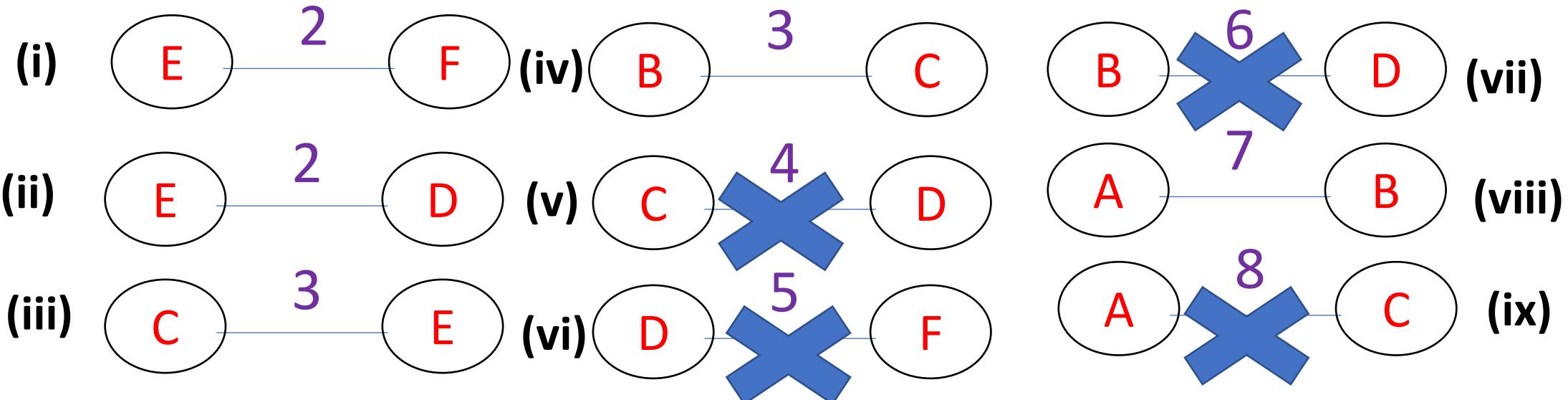
## Timing Analysis of Krushkal's Algorithm

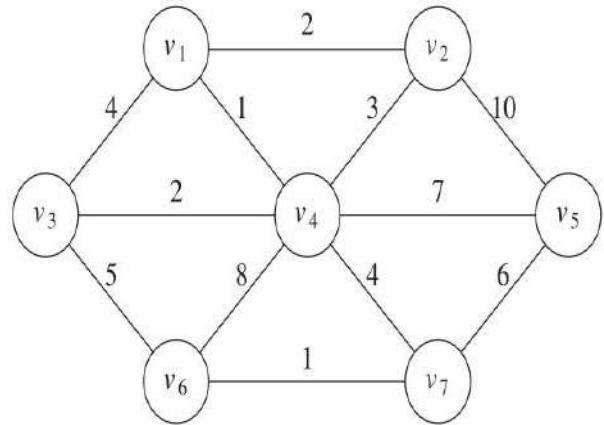
In **Kruskal's algorithm**, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be  $O(E \log V)$ , which is the overall Time Complexity of the algorithm.

# Minimum Cost Spanning Tree Krushkal's Algorithm

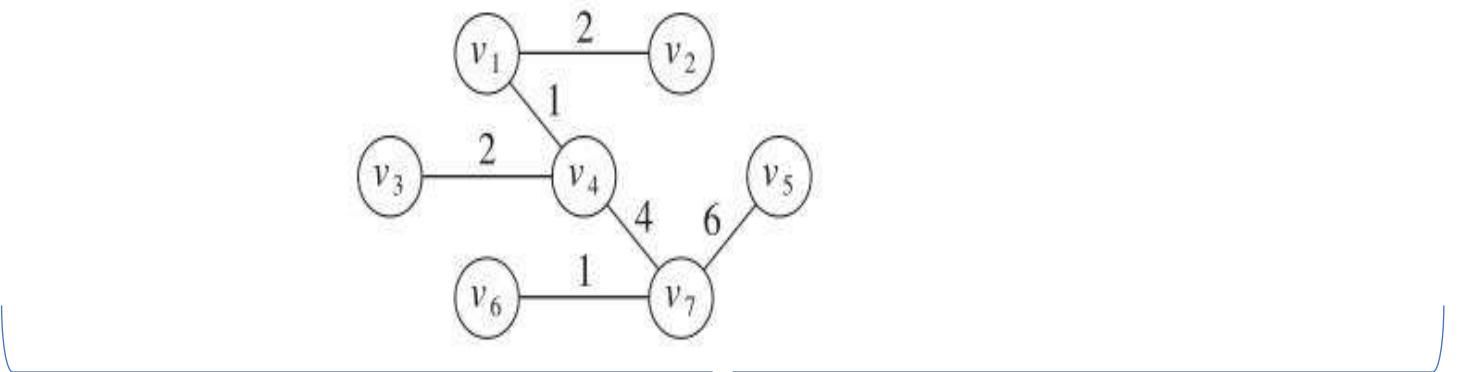
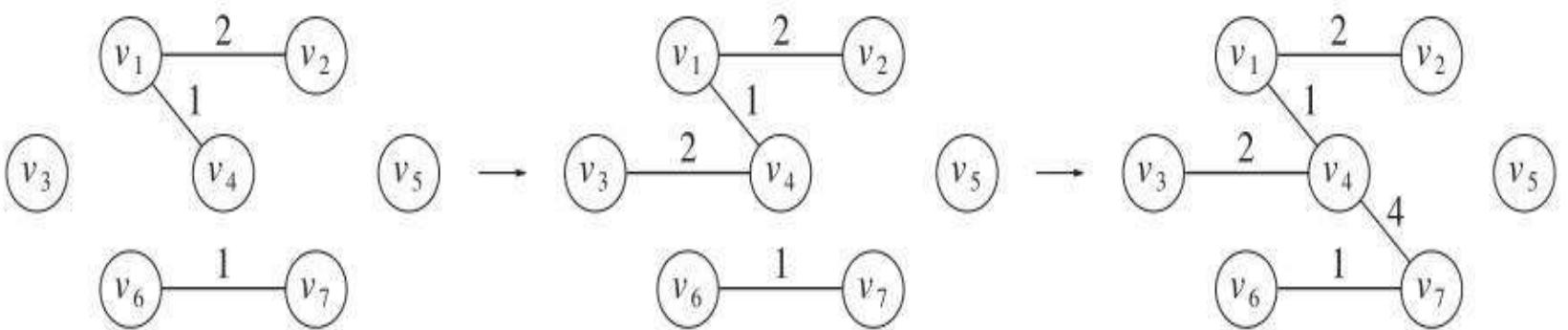
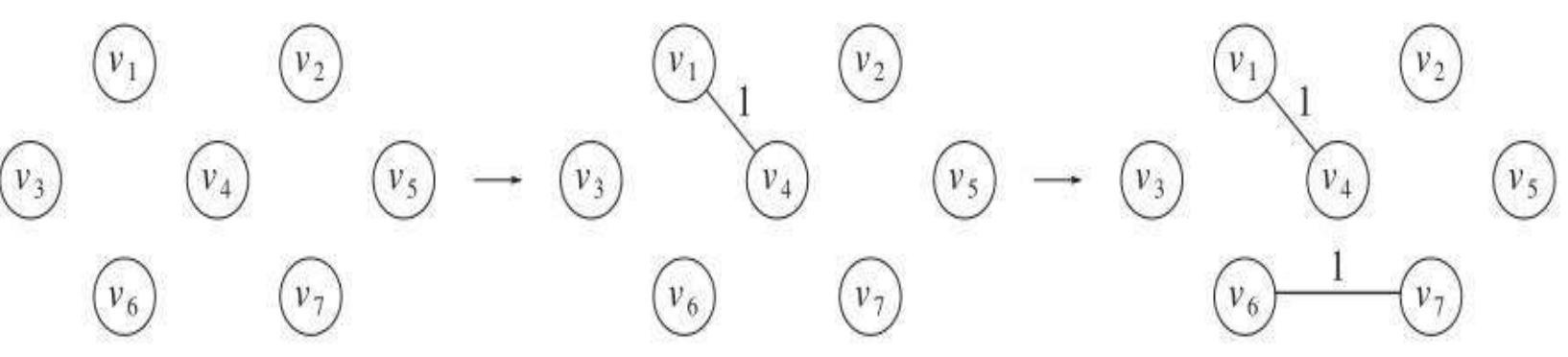


1. In the given graph G, delete all the loops and parallel edges. [ delete with maximum cost ]
2. Arrange all the edges in incremental order





Graph G



Spanning Trees

## Minimum Cost Spanning Tree: Prim's Algorithm

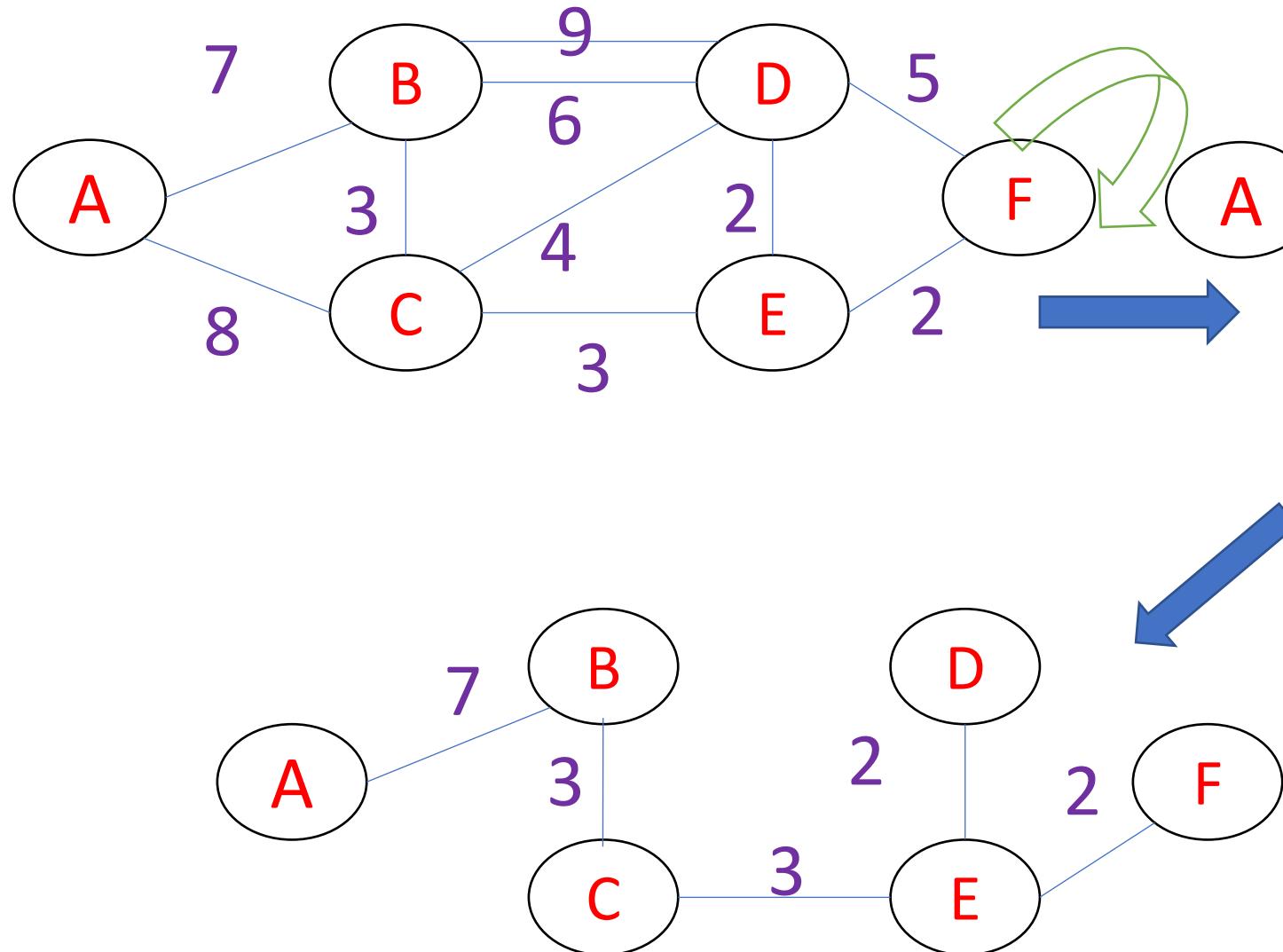
1. Remove loops and parallel edges [Keep min weight ]
2. Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
3. While adding new edge , select edge with minimum weight out of edges from already visited vertices ( no cycle allowed).
4. Stop at  $[n-1]$  edges.

Note: Step 3 can be Implemented using priority queue.

### Timing Analysis of Prim's Algorithm

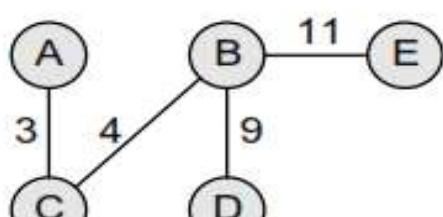
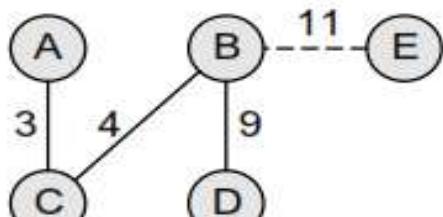
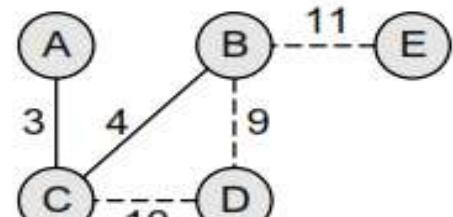
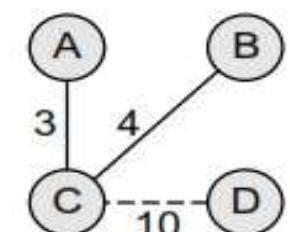
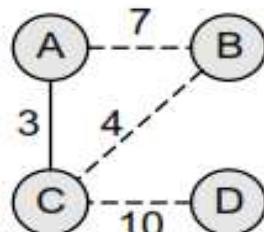
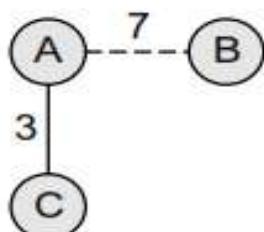
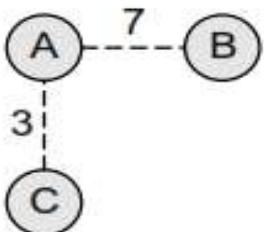
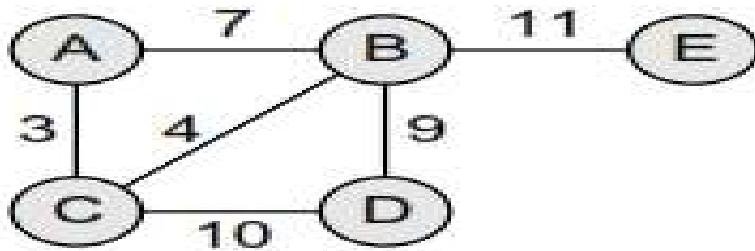
The time complexity of the Prim's Algorithm is  $O((V+E) * \log V)$  because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

# Minimum Cost Spanning Tree-Prim's Algorithm

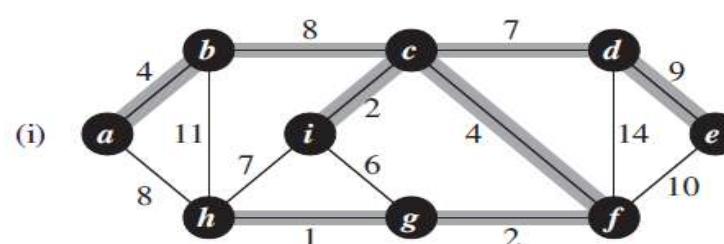
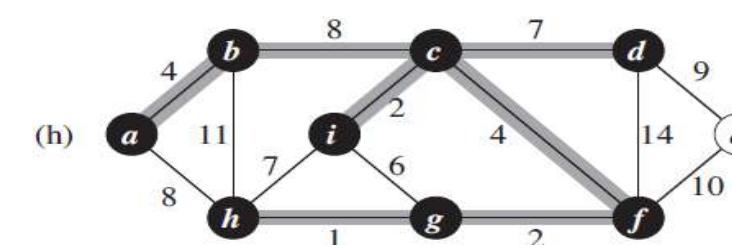
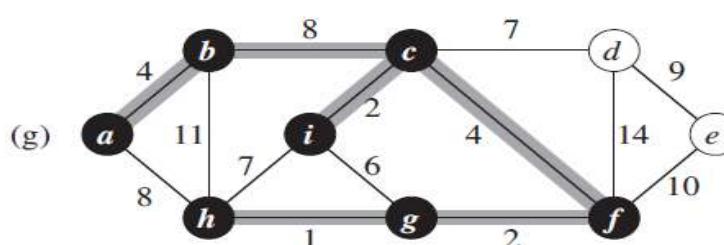
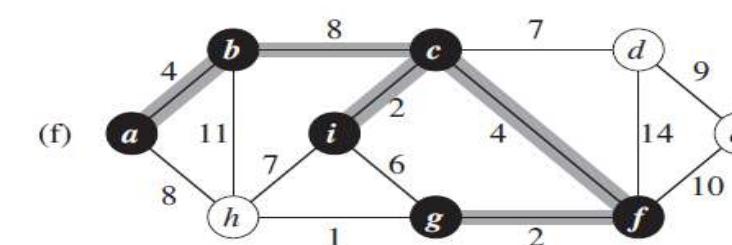
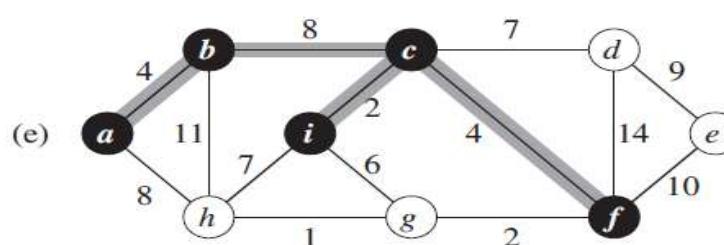
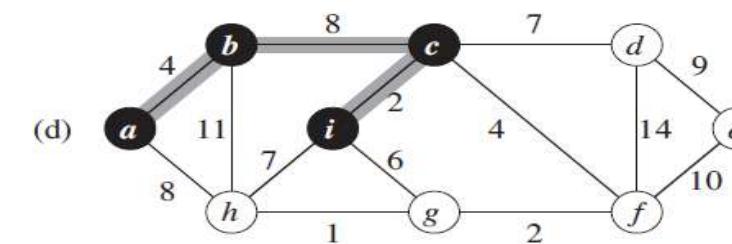
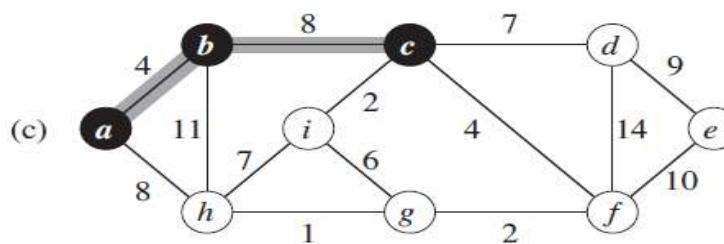
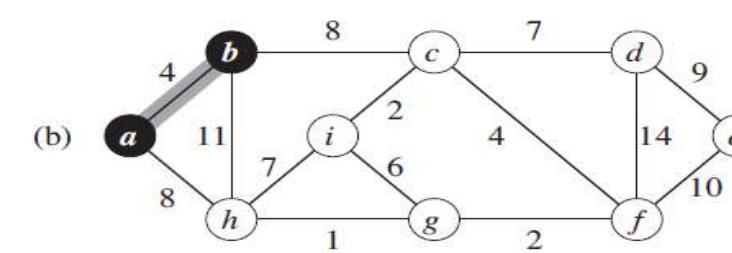
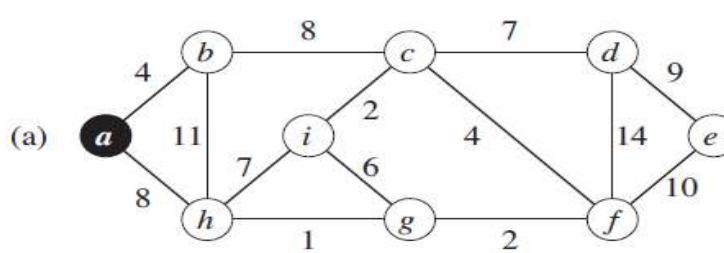


1. Remove loops and parallel edges [Keep min weight ]
2. While adding new edge , select edge with minimum weight out of edges from already visited vertices ( no cycle allowed).
3. Stop at  $[n-1]$  edges.

# MST using Prim's algorithm after each stage

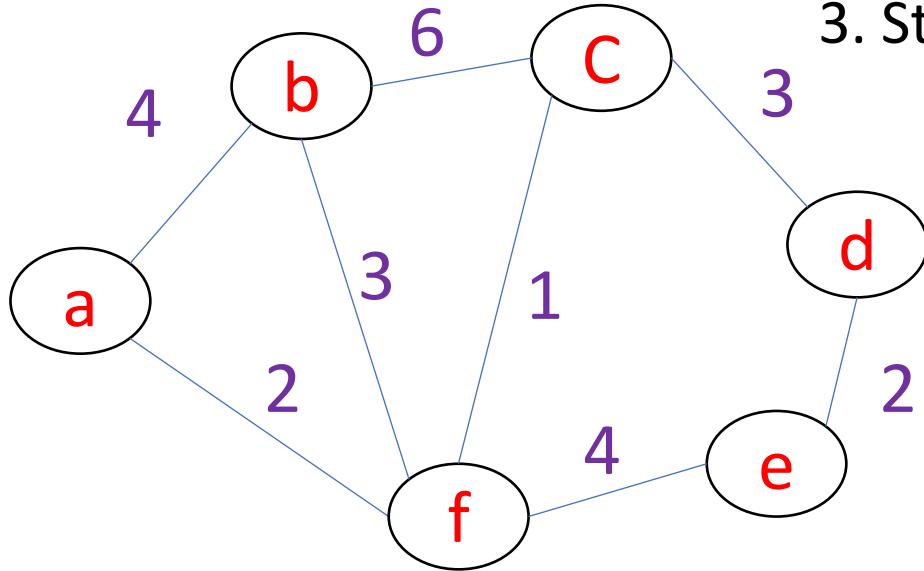


Cost=3+4+9+11=27



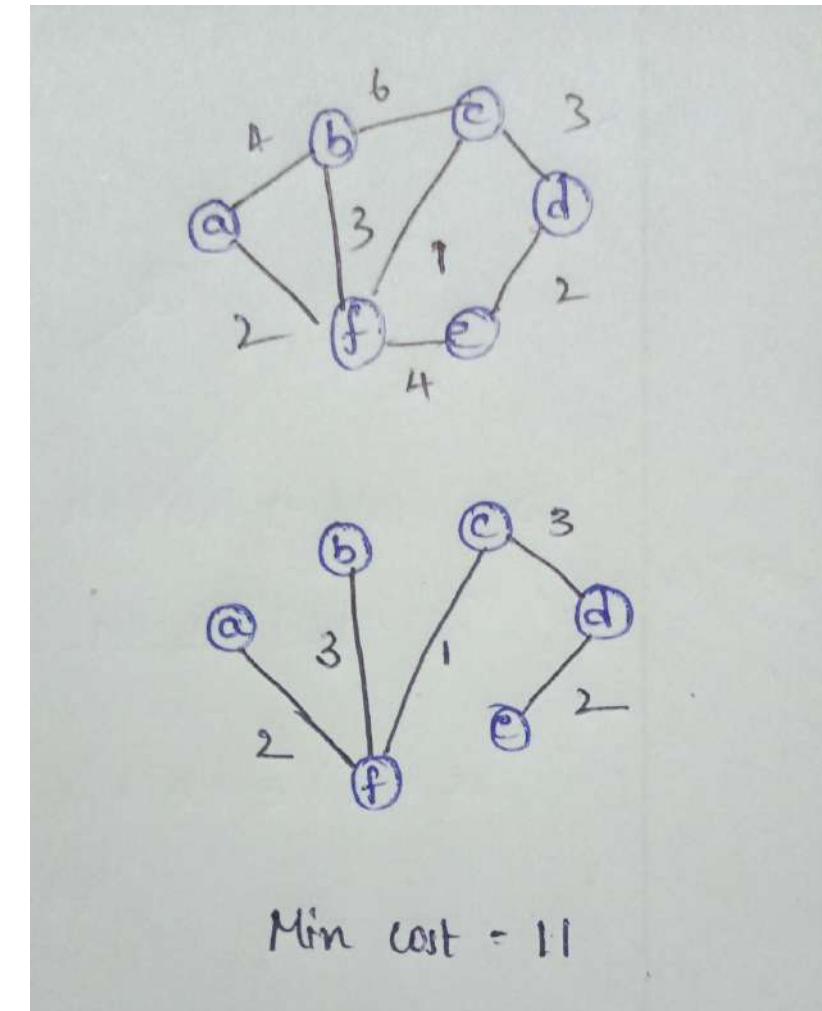
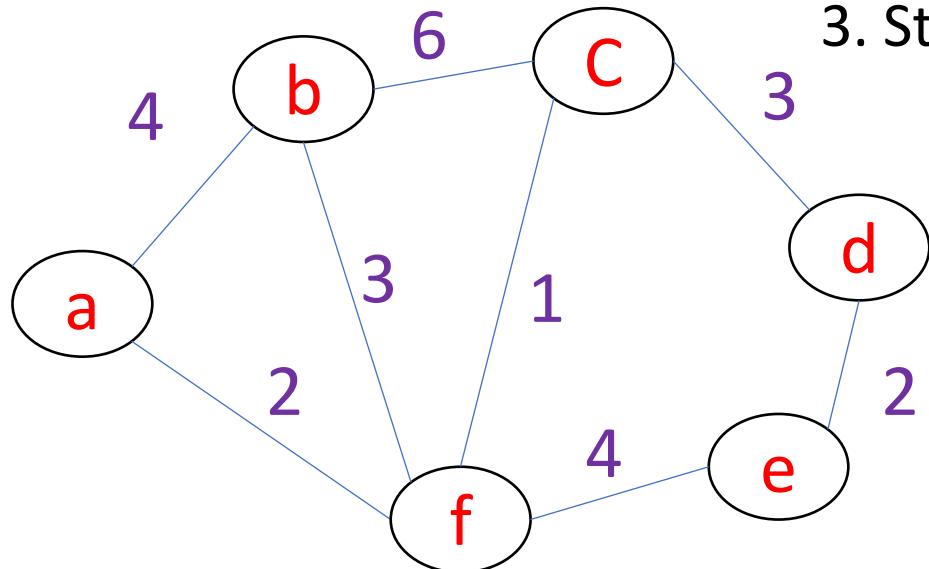
# Minimum Spanning Trees Algorithm- Prim's

1. Remove loops and parallel edges [Keep min weight ]
2. While adding new edge , select edge with minimum weight out of edges from already visited vertices ( no cycle allowed).
3. Stop at  $[n-1]$  edges.



# Minimum Spanning Trees Algorithm- Prim's

1. Remove loops and parallel edges [Keep min weight ]
2. While adding new edge , select edge with minimum weight out of edges from already visited vertices ( no cycle allowed).
3. Stop at  $[n-1]$  edges.



# **BCS202L- Data Structures and Algorithms**

**Dr.Priyanka N**

**Assistant Professor Senior Grade 1**

**School of Computer Science & Engineering**

**VIT,Vellore.**

# BCS202L- Data Structures and Algorithms

- Module-1: Algorithm Analysis
- Module-2: Linear Data Structures
- Module-3: Searching and Sorting
- Module-4: Trees
- Module-5: Graphs
- Module-6: Hashing
- Module-7: Heaps and AVL Trees

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# BCS202L- Data Structures and Algorithms

## Text Books:

1. Mark A. Weiss, Data Structures & Algorithm Analysis in C ++ , 4 th Edition, 2013, Pearson Education.

## Reference Books:

1. Alfred V. Aho, Jeffrey D. Ullman and John E. Hopcroft, Data Structures and Algorithms, 1983, Pearson Education.
2. Horowitz, Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, 2008, 2<sup>nd</sup> Edition, Universities Press.
3. Thomas H. Cormen, C.E. Leiserson, R L. Rivest and C. Stein, Introduction to Algorithms, 2009, 3<sup>rd</sup> Edition, MIT Press.

# Module 6: Hashing

## ● Hash functions

- Open Hashing( **Closed Addressing**)
  - Separate Chaining
- Closed Hashing(**Open Addressing**)
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

## ● Rehashing

## ● Extendible Hashing

# Rehashing

- When hash table becomes too full, running time for operations will take more time.
- **For Example:** Insertions might fail for closed hashing with quadratic resolution.
- To overcome this type of situation Rehashing technique is used.
- A solution, then, is to build another table that is about twice as big (**with an associated new hash function**) and scan down the entire original hash table, computing the new hash value for each (**non deleted**) element and inserting it in the new table.

# Rehashing

- Let us consider  $f(x) = X \% 7$  Then,

$X =$	13	15	6	24
$f(x) =$	6	1	0	3

Applying  
Linear  
probing

0	1	2	3	4	5	6
6	15		24			13

For Example : if 23 is inserted into the table, the resulting table will be almost full(70%), so new table will be created

0	1	2	3	4	5	6
6	15	23	24			13

# Rehashing

- **IDEA:** The size of this table is 17, because this is the first prime that is **twice as large as the old table size.**
- The new hash function is then  $h(x) = x \bmod 17$ . The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
						6	23	24					13		15	

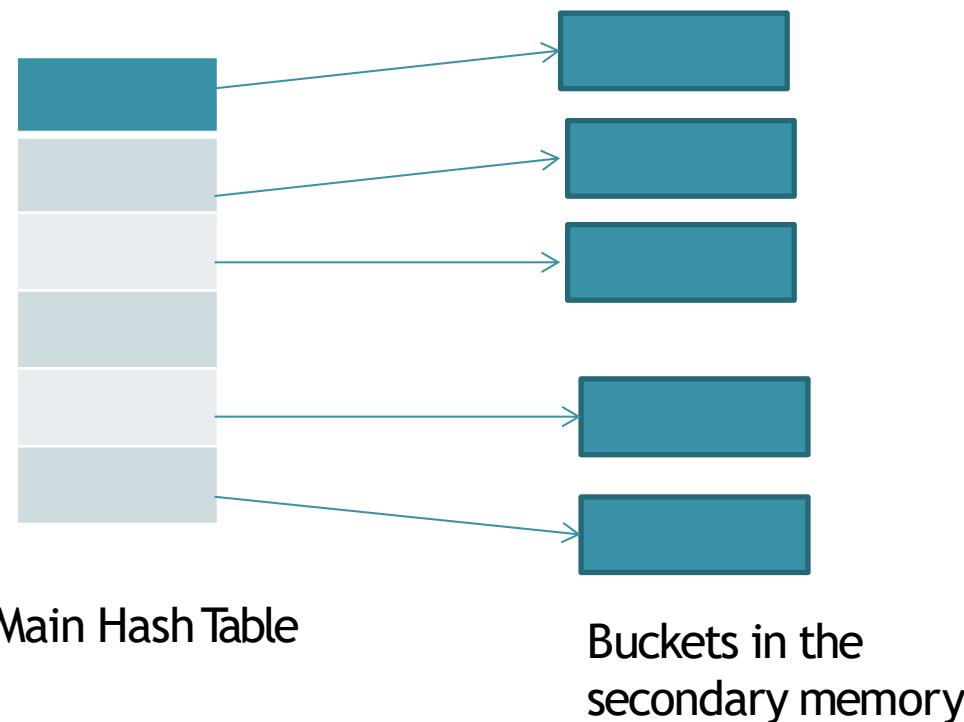
Linear probing

# Rehashing

- Go to rehashing ,
  - When table is half full.
  - When an insertion fails.
  - When table reaches load factor.
- Time Taking Operation
- Running time is  $O(N)$ , since there are  $N$  elements to rehash and the table size is roughly  $2N$

# Extendible hashing

- Main Hash table is stored in Main memory and the buckets are stored in the disk.
- Each value in the hash table is a pointer to a bucket in the secondary memory.
- Hash function is applied on the input value to generate the bucket pointer and perform the operations.

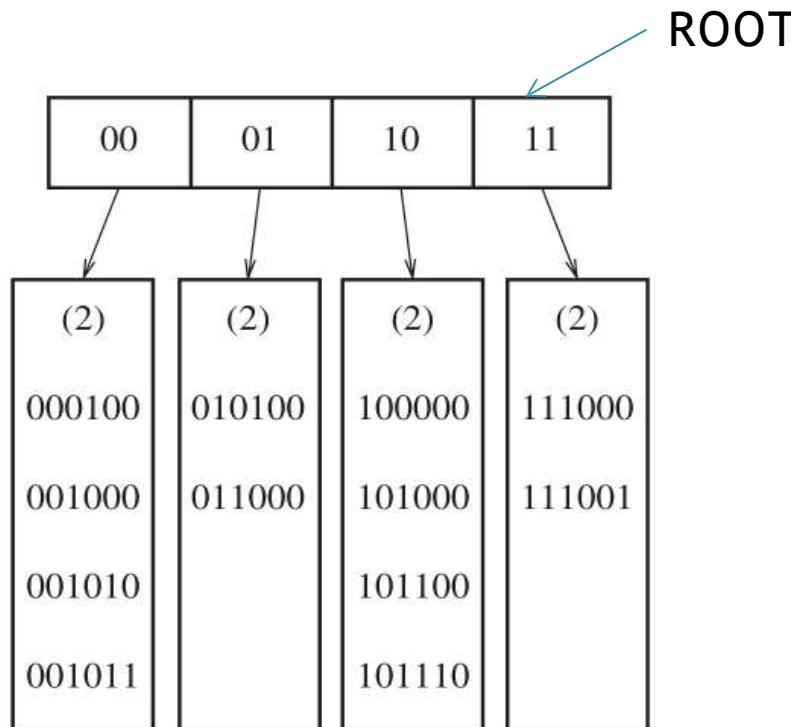


# Example

- Input our data consists of several 6-bit integers.
- To store these values??
- The root of the “tree” contains four pointers determined by the leading two bits of the data.

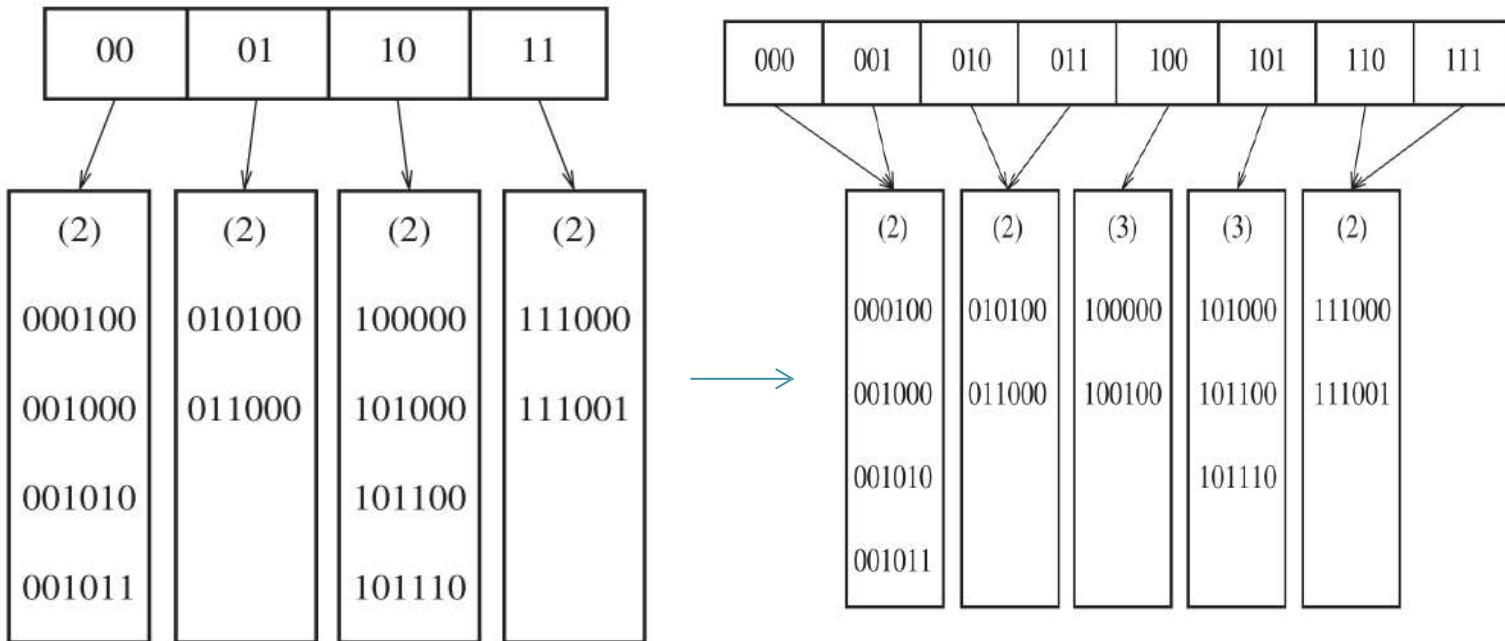
D- Number of bits used by the root. So number of entries in the leaf should be  $2^D$

Each leaf has up 4 to elements



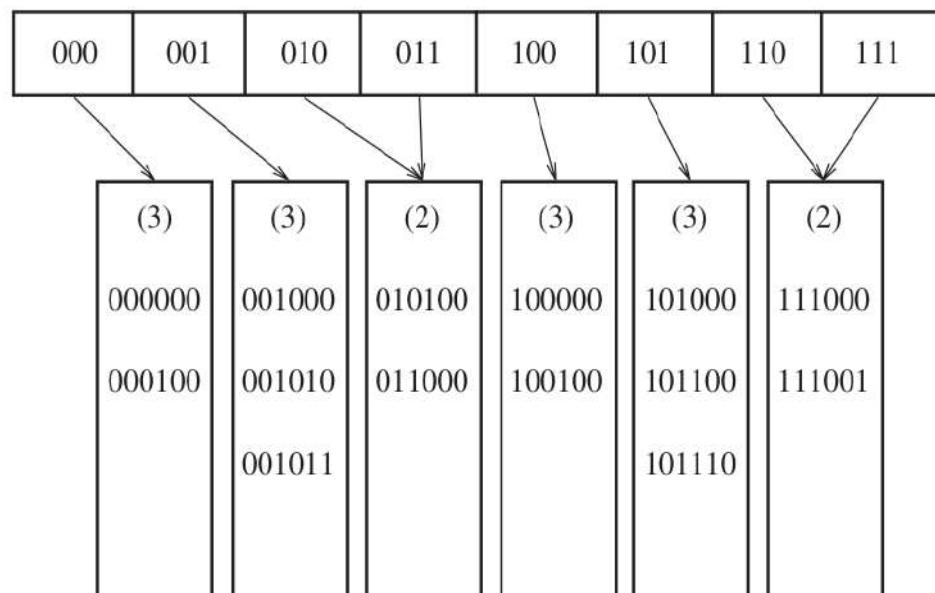
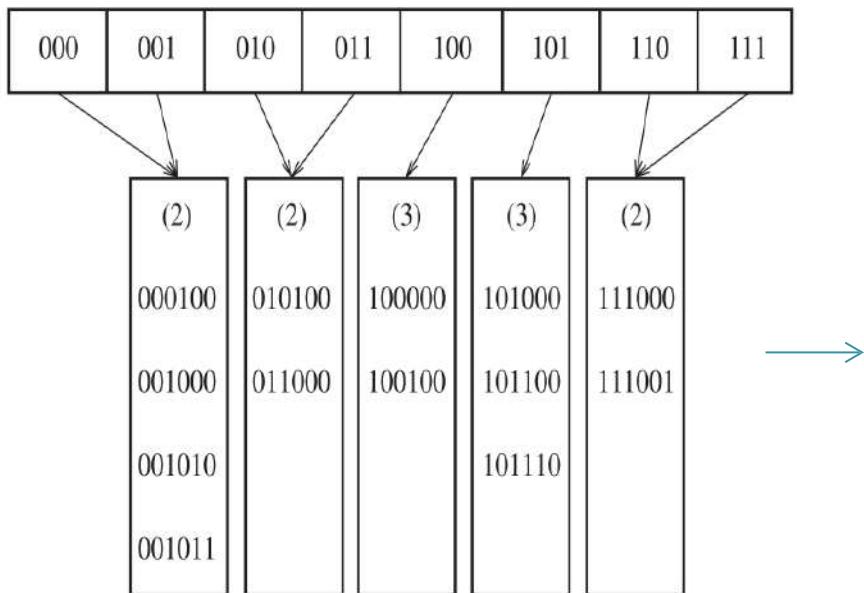
# Example

- Insert the key 100100, it points to third leaf, but it is already full, so split this leaf into two leaves, which are determined by first three bits.



# Example

- Insert the key 000000, it points to first leaf, but it is already full, so split this leaf into two leaves, which are determined by first three bits.



# Extendible hashing

## ● Advantages

- Simple
- Quick access time for insert and find operations in large databases.

## ● Disadvantages

- Not suitable when more duplicates are taken

# HEAPS AND AVL TREES

Dr. Priyanka N  
Assistant Professor Senior Grade 1  
School of Computer Science & Engineering  
VIT, Vellore.

<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# HEAPS AND AVL TREES

- Introduction to Heaps
- Heap Sort
- Priority Queue using Heaps
- AVL Trees
  - Terminologies
  - Basic Operations
    - Rotation
    - Insertion
    - Deletion

# AVL Trees

- AVL tree is a height-balanced binary search tree
- It is also a binary search tree but it is a balanced tree.
- **What is Balanced Tree?**
  - A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1
- In an AVL tree, every node maintains an extra information known as **balance factor**

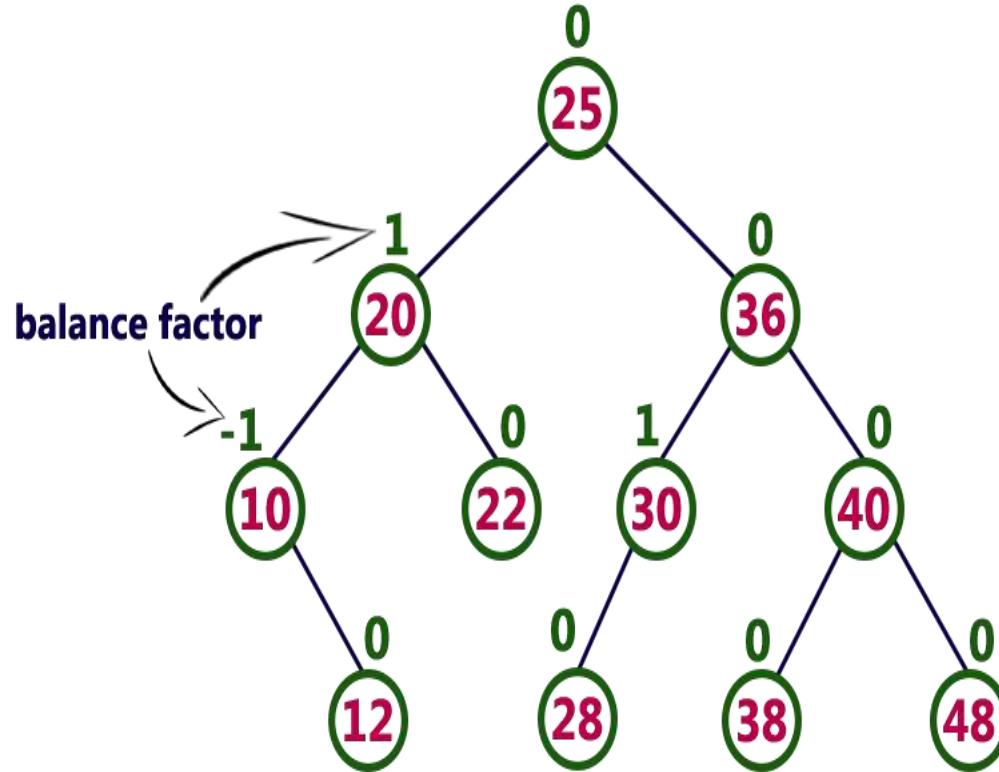
# AVL Trees

- AVL tree was introduced by G.M. Adelson-Velsky and E.M. Landis
- Balance factor of a node is the difference between the heights of the left and right subtrees of that node.
- Balance factor = **height of left subtree - height of right subtree**

# Why AVL Trees?

- AVL tree controls the height of the binary search tree by not letting it to be skewed.
- Time taken for all operations in BST =  $O(h)$
- When it is Skewed it is  $O(n)$ , For limiting this, AVL tree is Introduced.
- For example : Searching is inefficient(i.e takes more time) in BST, When large number of nodes available in the tree when height is not balanced

# Example of AVL Trees



Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

# AVL Tree Rotations

- When performing operations like insertion and deletion - check the **balance factor** of every node in the tree.
- **If not Balanced, What should be done?**
  - **Rotation** operations is done to make the tree balanced
- **What is Rotation??**
  - Rotation is the process of moving nodes either to left or to right to make the tree balanced.

# Classification of Rotations

## 1. Single Rotation

1. Left Rotation(LL Rotation)
2. Right Rotation(RR Rotation)

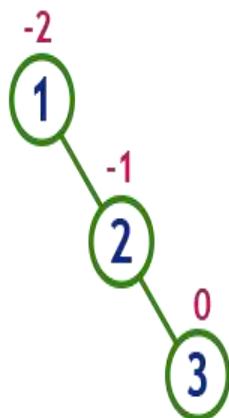
## 2. Double Rotation

1. Left Right Rotation(LR Rotation)
2. Right Left Rotation(RL Rotation)

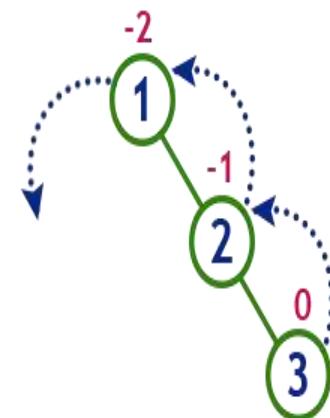
# Single Left Rotation (LL Rotation)

- Every node moves one position to left from the current position

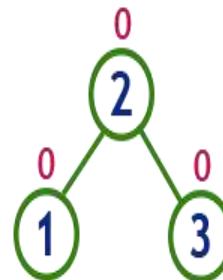
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use  
LL Rotation which moves  
nodes one position to left

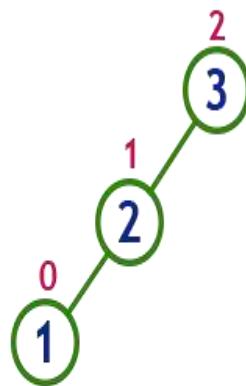


After LL Rotation  
Tree is Balanced

# Single Right Rotation (RR Rotation)

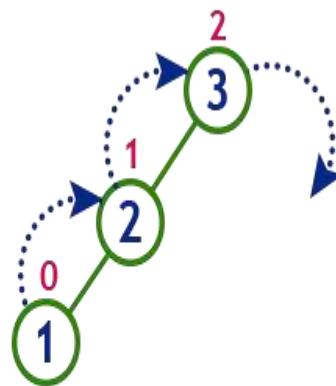
- Every node moves one position to right from the current position

insert 3, 2 and 1

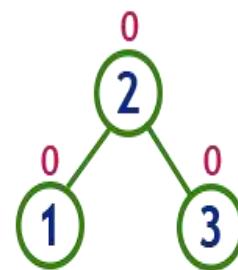


Tree is imbalanced

because node 3 has balance factor 2



To make balanced we use  
RR Rotation which moves  
nodes one position to right

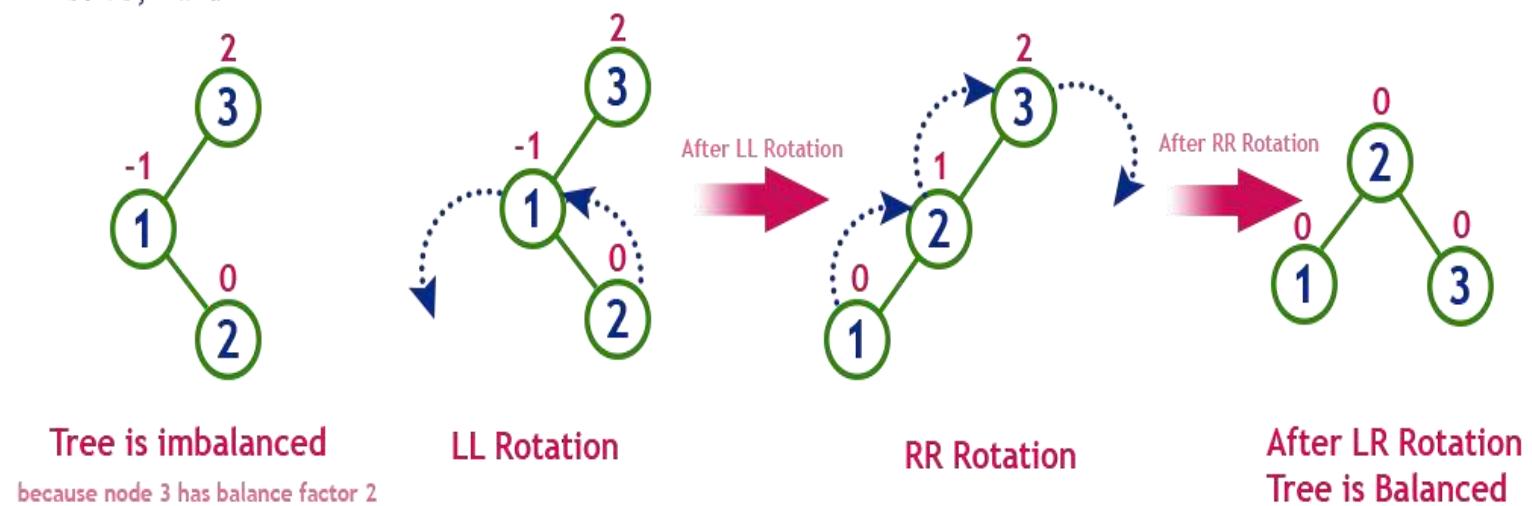


After RR Rotation  
Tree is Balanced

# Double Rotation- Left Right Rotation (LR Rotation)

- Sequence of single left rotation followed by a single right rotation
- Every node moves one position to the left and one position to right from the current position

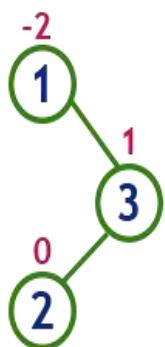
insert 3, 1 and 2



# Double Rotation- Right Left Rotation (RL Rotation)

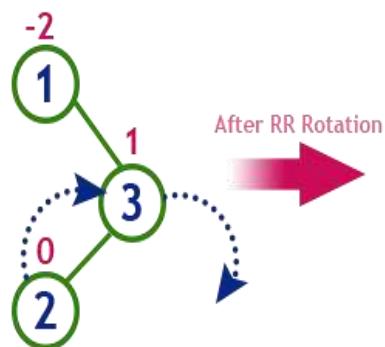
- Sequence of single right rotation followed by a single left rotation
- Every node moves one position to the right and one position to left from the current position

insert 1, 3 and 2

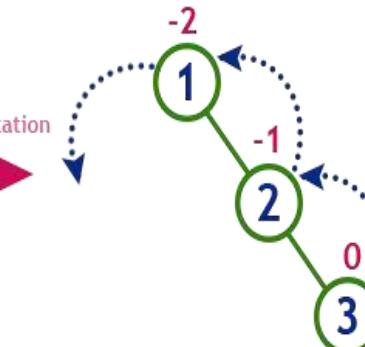


Tree is imbalanced

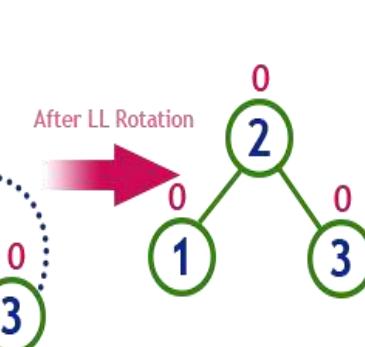
because node 1 has balance factor -2



After RR Rotation



LL Rotation



After RL Rotation  
Tree is Balanced

# Operations on AVL Tree

- Insertion
- Search
- Deletion

# Insertion Operation

- As in BST, a new node is always inserted as a leaf node.
- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the **Balance Factor of every node.**
- Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
- Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

# Construction of AVL Tree (1 to 8)

- Insert 1

$BF=0$



*TREE is Balanced*

- Insert 2

$BF=-1$



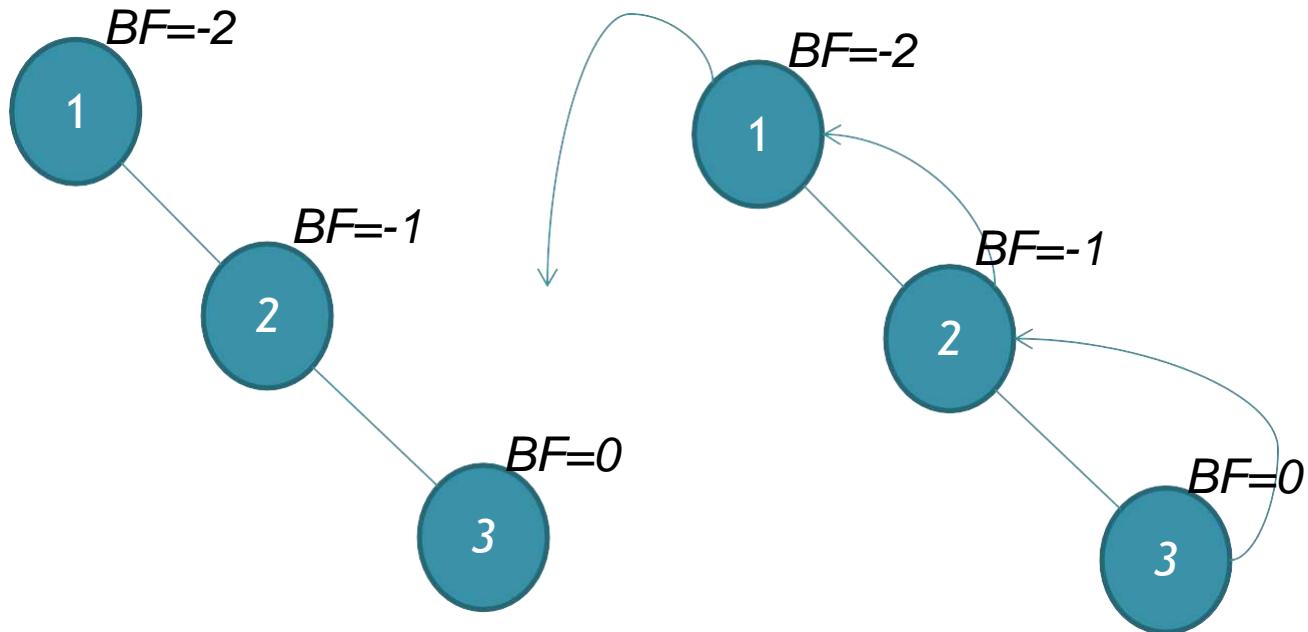
$BF=0$



*TREE is Balanced*

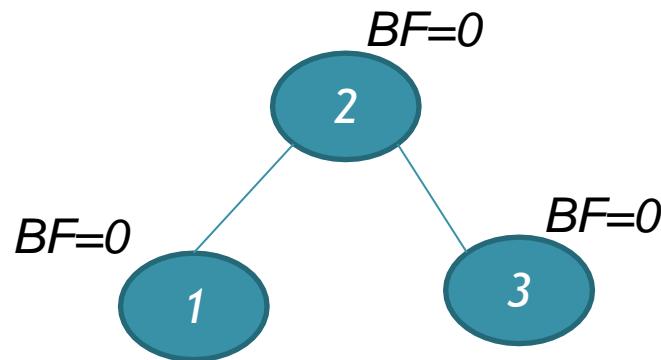
# Construction of AVL Tree (1 to 8)

## ● Insert 3



*TREE is imbalanced*

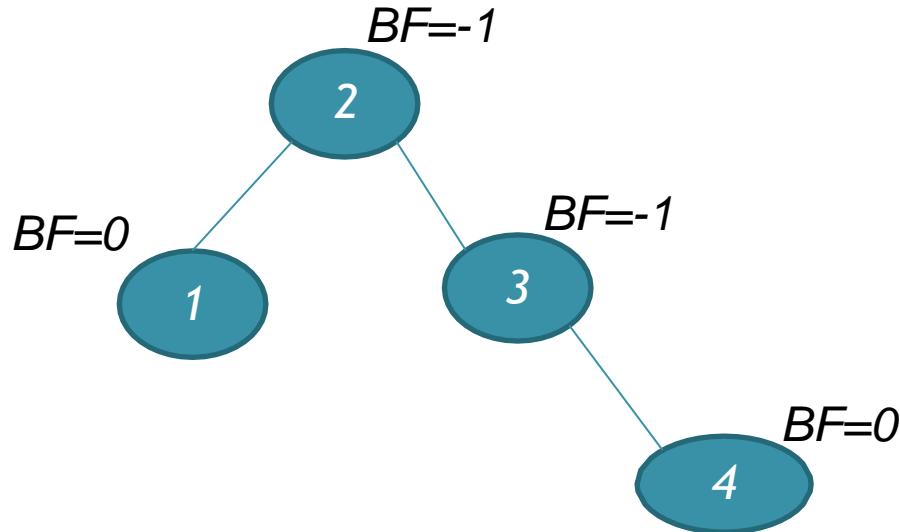
*LL ROTATION*



*TREE is balanced*

# Construction of AVL Tree (1 to 8)

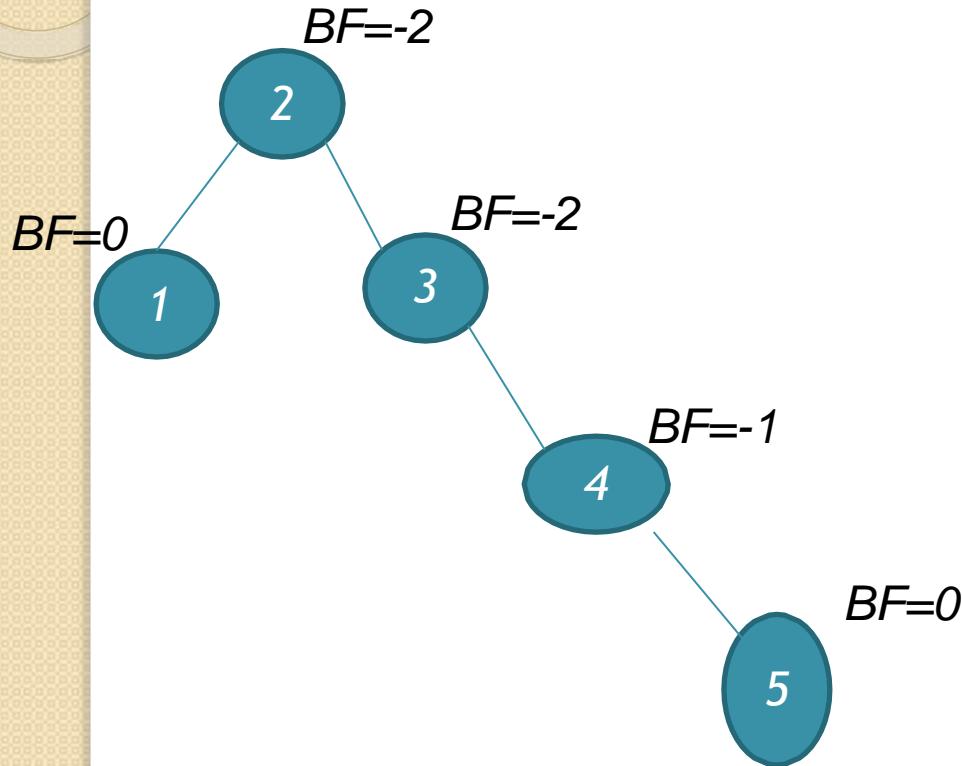
- Insert 4



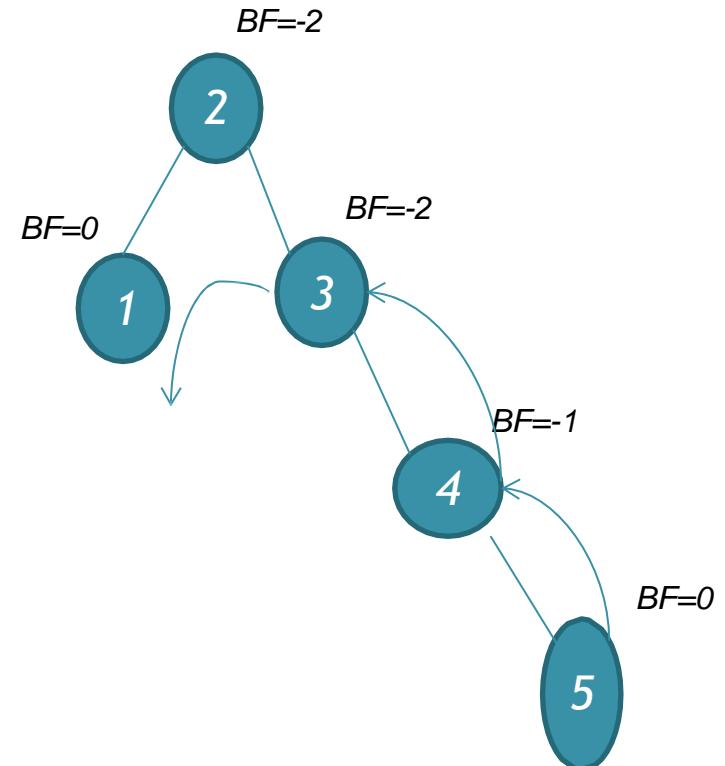
*TREE is balanced*

# Construction of AVL Tree (1 to 8)

- Insert 5



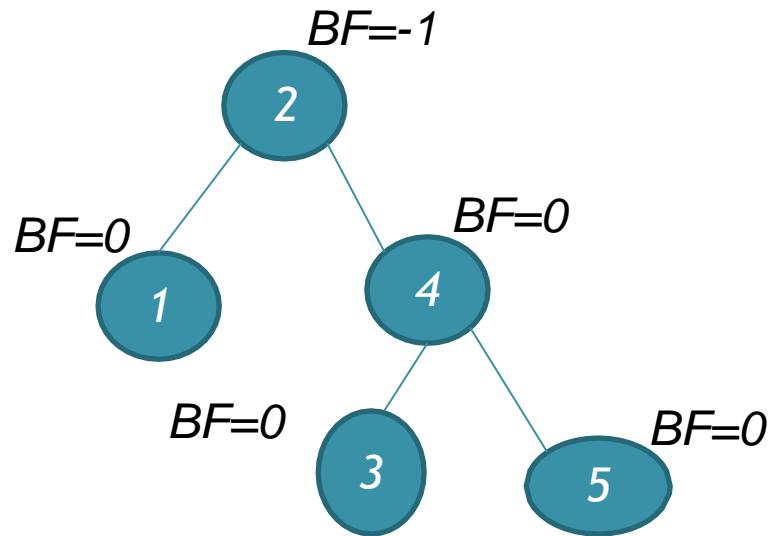
*TREE is imbalanced*



**LL ROTATION AT 3**

# Construction of AVL Tree (1 to 8)

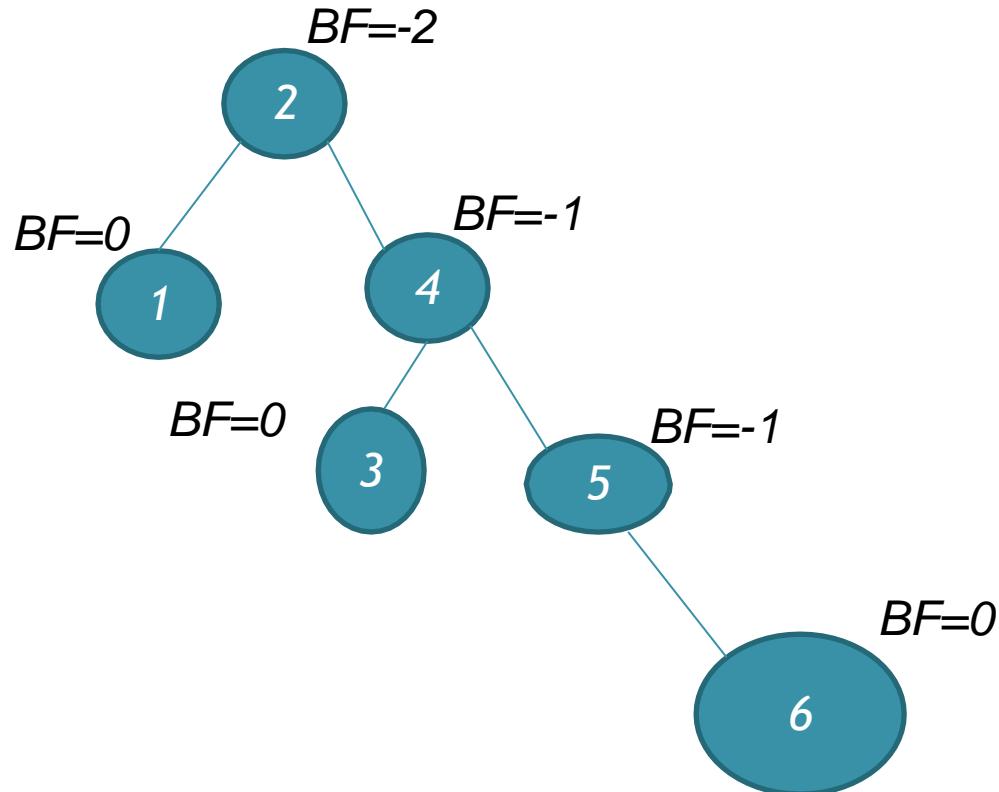
- Result of Inserting 5



*TREE is balanced*

# Construction of AVL Tree (1 to 8)

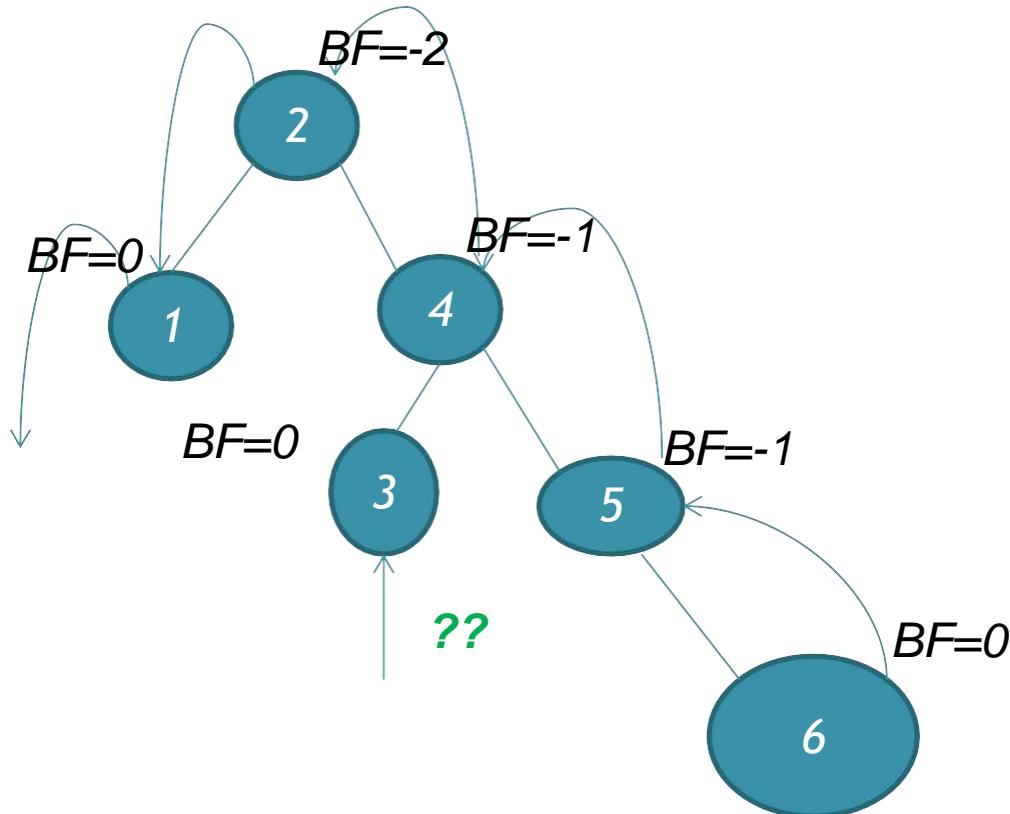
- Insert 6



*TREE is imbalanced*

# Construction of AVL Tree (1 to 8)

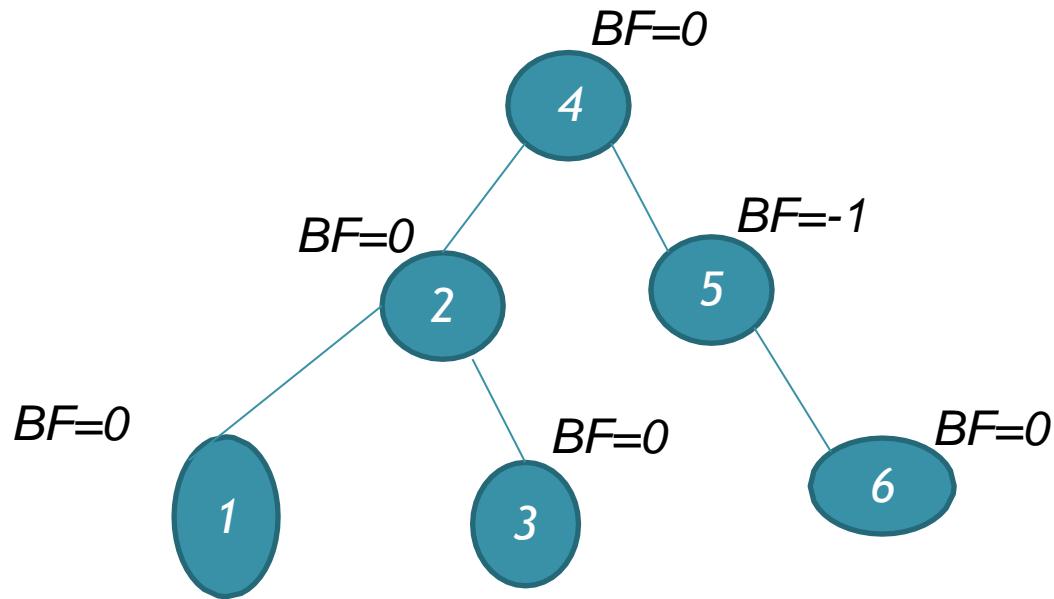
- Rotation process at Inserting 6



*LL Rotation at 2*

# Construction of AVL Tree (1 to 8)

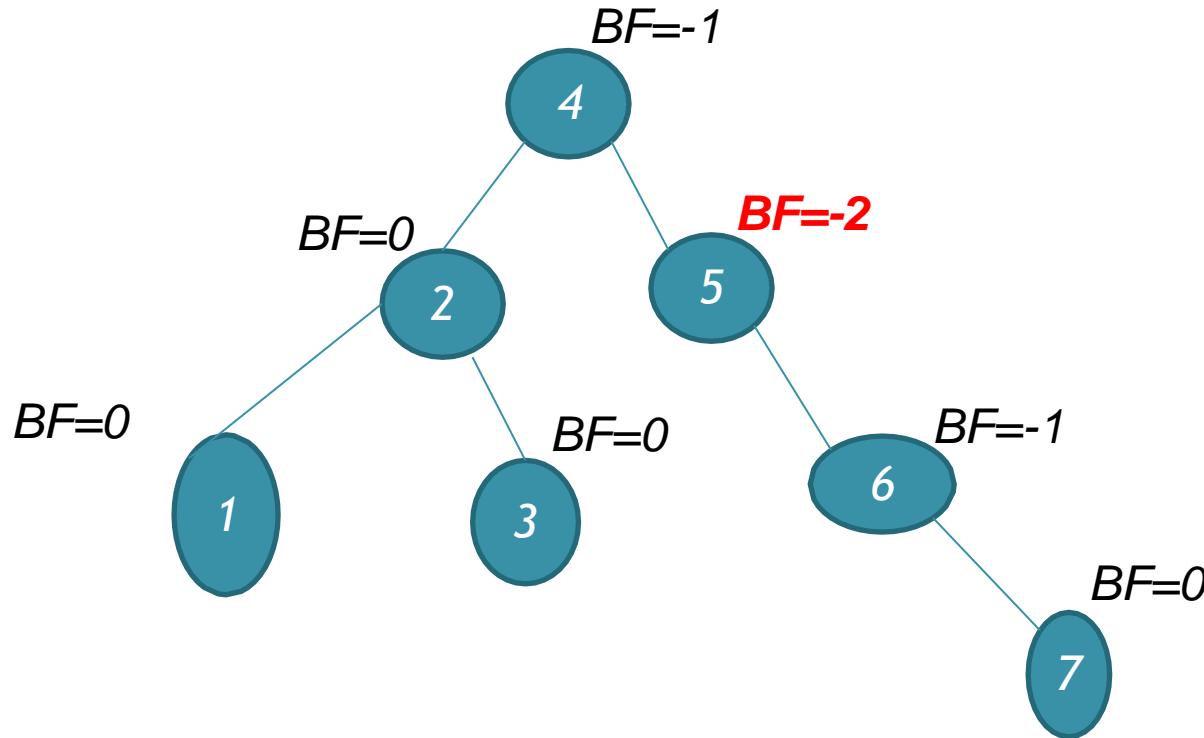
- Result of Inserting 6



*TREE is balanced*

# Construction of AVL Tree (1 to 8)

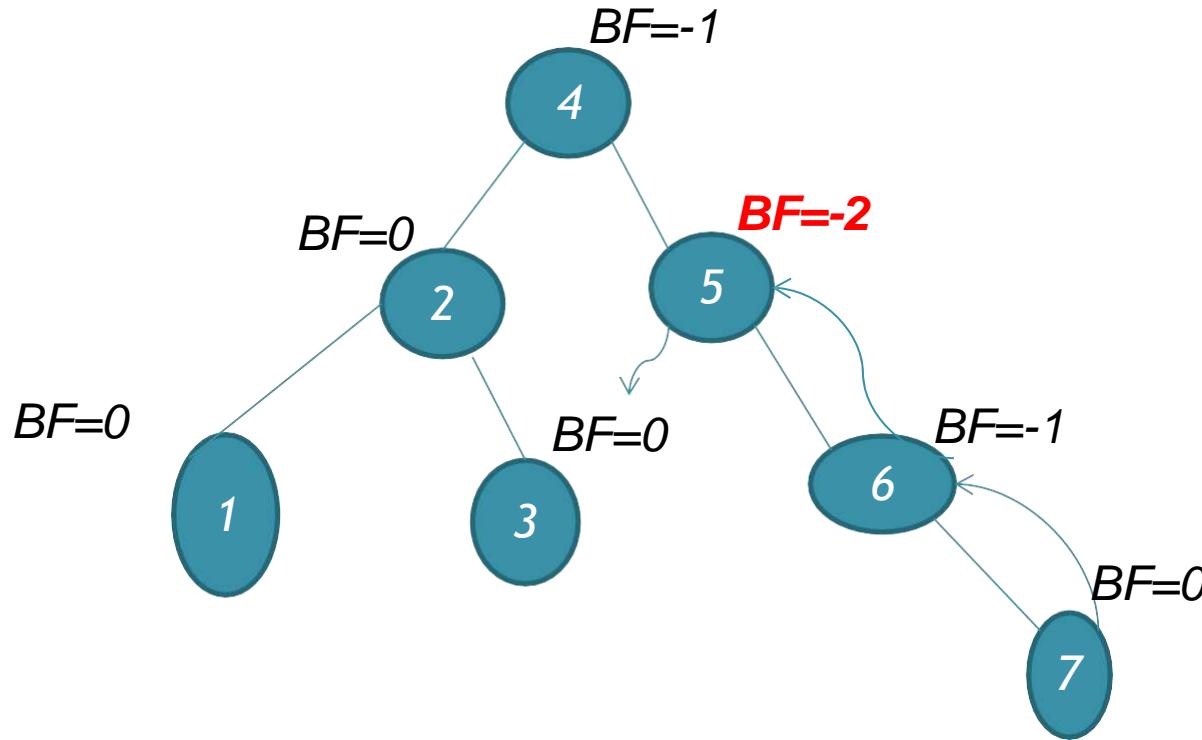
- Insert 7



*TREE is imbalanced*

# Construction of AVL Tree (1 to 8)

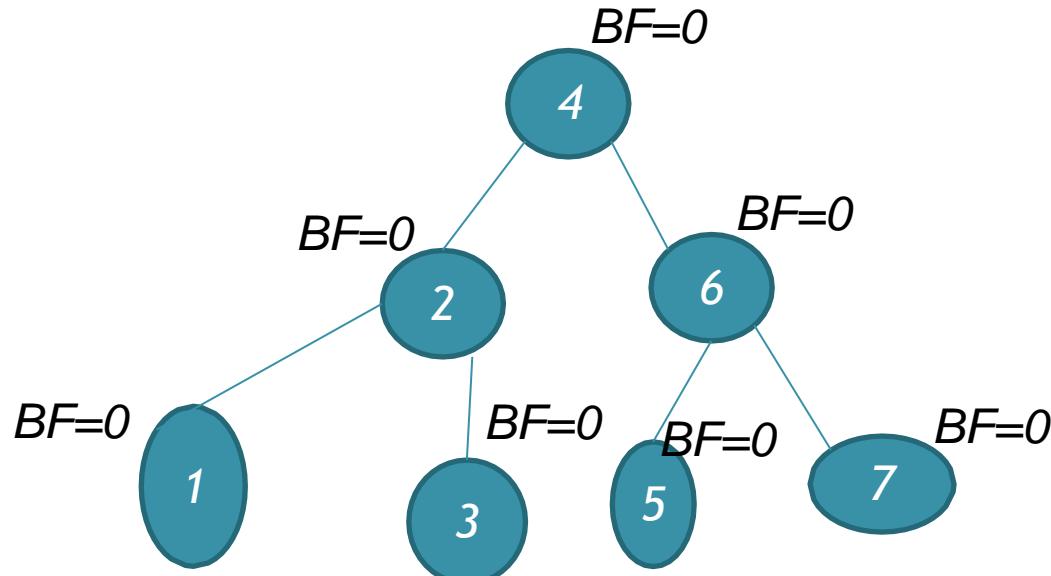
- Rotation Process when Inserting 7



*LL Rotation at 5*

# Construction of AVL Tree (1 to 8)

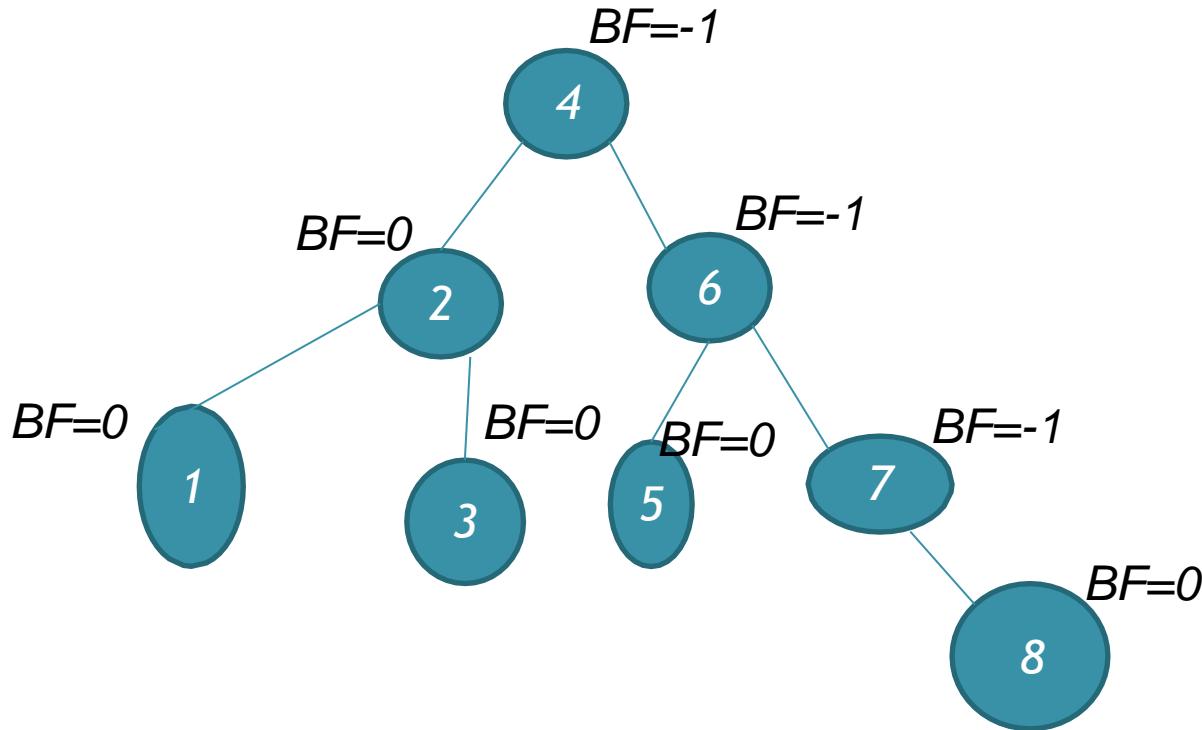
- Result of Inserting 7



*TREE is balanced*

# Construction of AVL Tree (1 to 8)

- Insert 8



*TREE is balanced*

# Deletion Operation

- Similar to deletion operation in BST
- Every deletion operation, we need to check with the Balance Factor condition
- If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

# **IMPLEMENTATION OF AVL TREES**

**Dr.Priyanka N**

**Assistant Professor Senior Grade 1**

**School of Computer Science & Engineering**

**VIT,Vellore.**

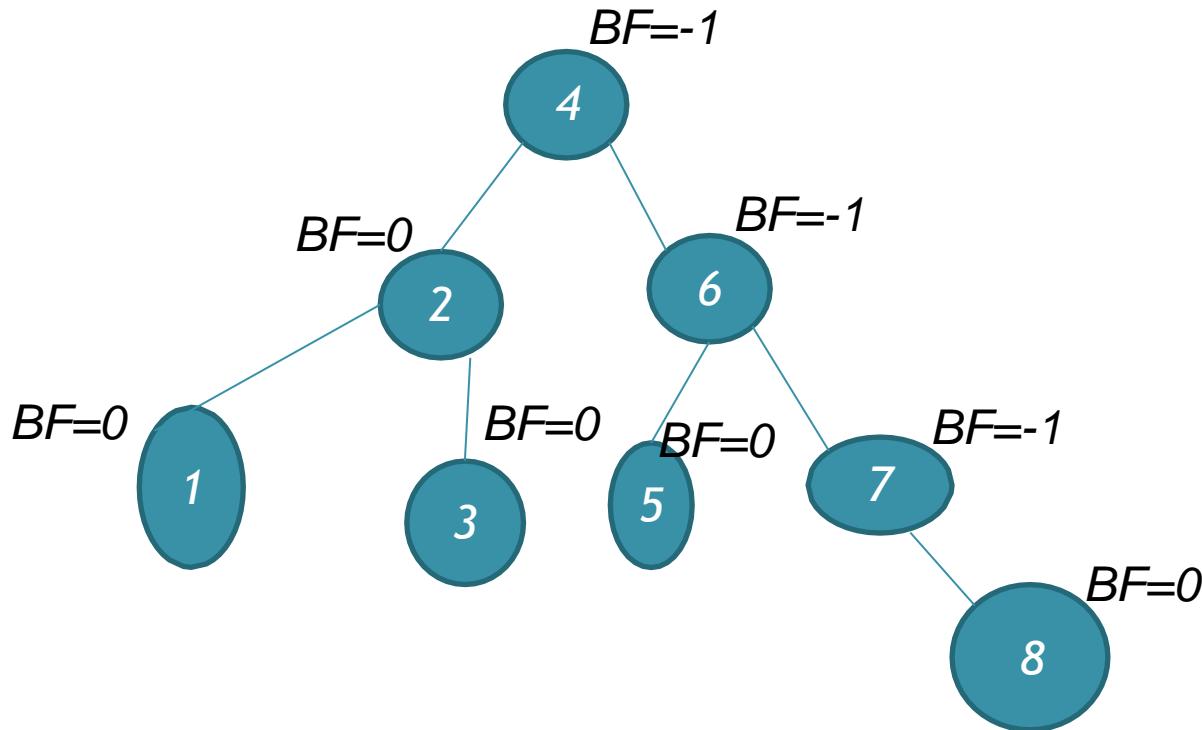
<b>Module:1</b>	<b>Algorithm Analysis</b>	<b>8 hours</b>
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
<b>Module:2</b>	<b>Linear Data Structures</b>	<b>7 hours</b>
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
<b>Module:3</b>	<b>Searching and Sorting</b>	<b>7 hours</b>
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
<b>Module:4</b>	<b>Trees</b>	<b>6 hours</b>
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the $k^{\text{th}}$ minimum element.		
<b>Module:5</b>	<b>Graphs</b>	<b>6 hours</b>
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
<b>Module:6</b>	<b>Hashing</b>	<b>4 hours</b>
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
<b>Module:7</b>	<b>Heaps and AVL Trees</b>	<b>5 hours</b>
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

# AVL Trees

- AVL tree is a height-balanced binary search tree
- It is also a binary search tree but it is a balanced tree.
- **What is Balanced Tree?**
  - A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1
- In an AVL tree, every node maintains an extra information known as **balance factor**

# Construction of AVL Tree (1 to 8)

Example:



*TREE is balanced*

# Node Structure

Struct Node

{

int data;

struct Node \*left;

struct Node \*right;

int height;

};



**NODE**

# Height of the Node

```
height(Struct Node *Node)
{
    if(Node==NULL)
        return 0;
    return Node->Height;
}
```



# Creation of Node

Create newNode(int value)

{

```
Struct Node *newnode;
newnode= malloc(sizeof(Struct node));
newnode->data=value;
newnode->left=NULL;
newnode->right=NULL;
newnode->height=1;
```

return newNode;

}

NULL	10	NULL	1
------	----	------	---

*newNODE*

# Balance Factor of the Node

Balance Factor(Struct Node \*Node)

```
{
```

```
    if(Node==NULL)
```

```
        return 0;
```

```
    return height(Node->left)-height(Node->right) ;
```

```
}
```



# Left Rotate

Struct Node \*Left Rotate( Struct node \*x)

{

    Struct Node \*y= x->right;

    Struct Node \*T= y->left;

    y->left =x;

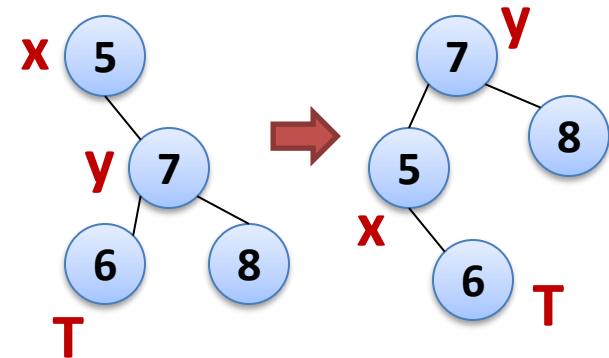
    x->right=T;

    x->height= max(height(x->left), height(x->right))+1;

    y->height=max(height(y->left),height(y-> right))+1;

    return y;

}



# Right Rotate

Struct Node \*Right Rotate( Struct node \*y)

{

    Struct Node \*x= y->left;

    Struct Node \*T= x->right;

    x->right=y;

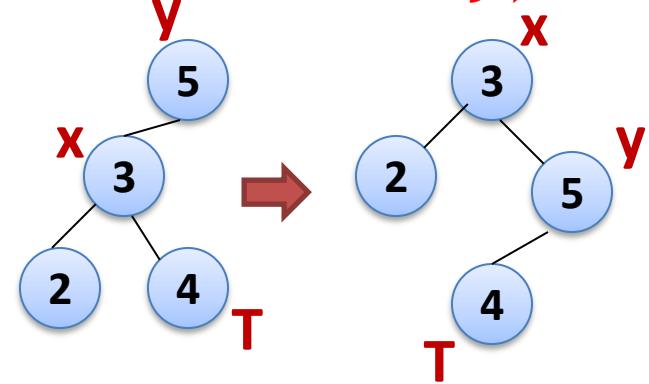
    y->left =T;

    x->height= max(height(x->left), height(x->right))+1;

    y->height= max(height(y->left), height(y->right))+1;

    return x;

}



# Finding Max

```
Max( a,b)
```

```
{
```

```
    return(a>b)? a:b;
```

```
}
```

//Returns which is maximum among a and b

# Insertion Operation

```
insert(struct * node, int value)
```

```
{
```

```
    if(node == NULL)
```

```
        return newNode(value);
```

```
    if(value < node->data)
```

```
{
```

```
        node->left = insert(node->left, value);
```

```
    else if(value > node->data)
```

```
        node->right = insert(node->right, value);
```

```
    else
```

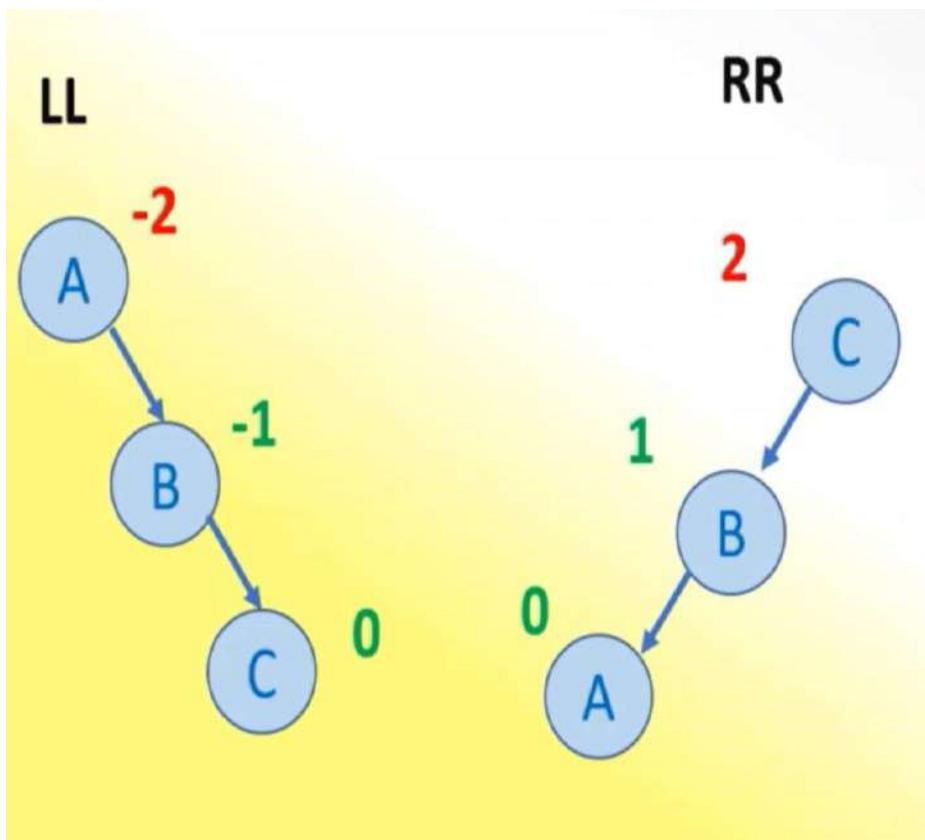
```
        return node
```

```
}
```

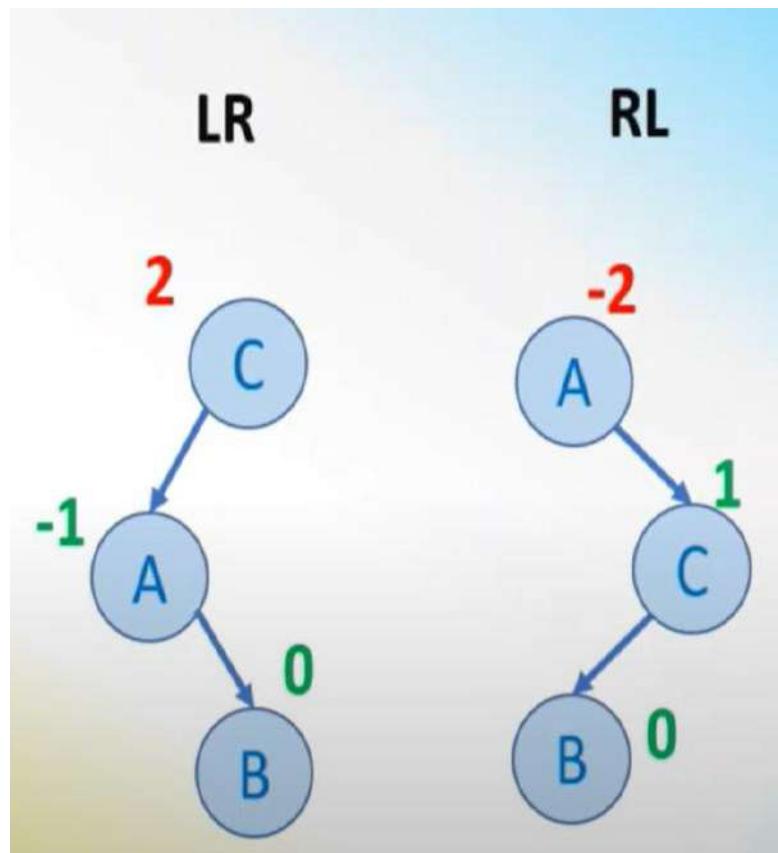
*Node and value to be inserted is passed every time*

```
// finds the correct position and inserts node
```

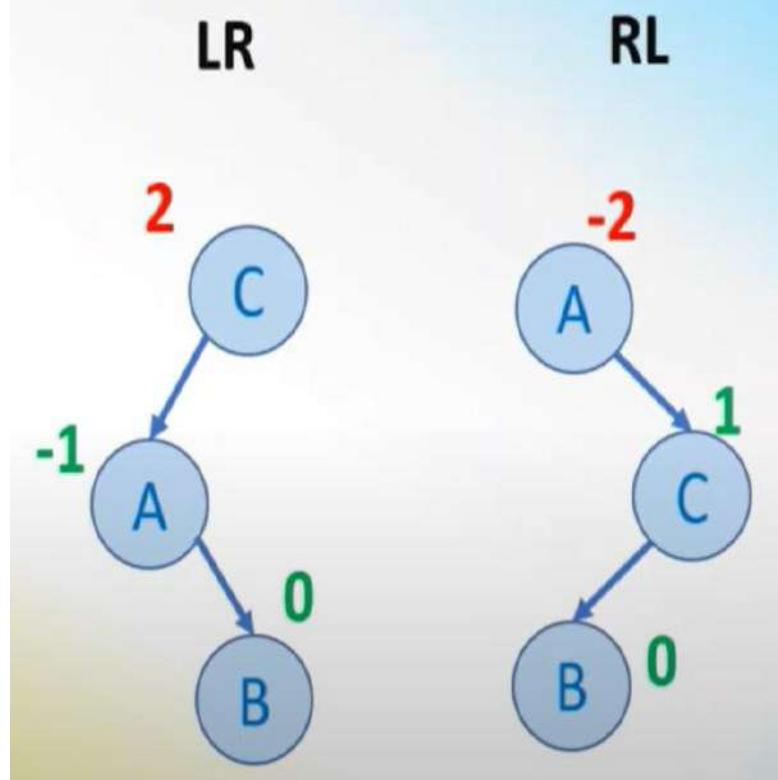
## ROTATIONS



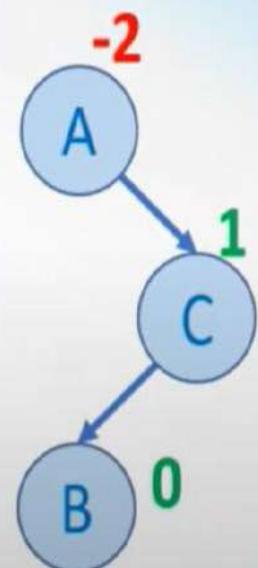
RR



LR



RL



# Insertion Operation

```
//Update the height & Balance Factor of each node
Node->height =max[height(node->left),height(node->right)]+1;
Balance = Balance Factor(node);

// ROTATIONS
if(balance>1 && value<node->left->value)// RR
    Rotation return rightrotate(node);
if(balance <-1&& value>node->right->value)// LL Rotation
    return leftrotate(node);
if(balance >1&& value>node->left->value)// LR Rotation
    node->left= leftrotate(node->left);
    return rightrotate(node);
if(balance <-1&& value<node->right->value)// RL Rotation
    node->right= rightrotate(node->right);
    return leftrotate(node);

return node;
}
```

```
deleteNode(struct node* root, int value)
{
    if (root == NULL) //value not found
        return root;
    //key less than value of root then it lies in left subtree
    if (value < root->data)
        root->left = deleteNode(root->left, value);
    //key greater than value of root then it lies in right subtree
    else if (value > root->data)
        root->right = deleteNode(root->right, value);
    //key is same as value of root, node be deleted is found
    else
    {
```

**// node with only one child or no child**

```
if (root->left == NULL)
{
    struct node* temp = root->right;
    free(root);
    return temp;
}

else if (root->right == NULL)
{
    struct node* temp = root->left;
    free(root);
    return temp;
}
```

```
// node with two children, Get smallest in  
// the right subtree)  
struct node* temp = minValueNode(root->right);  
root->value = temp->value;  
// Delete the node  
root->right = deleteNode(root->right, temp->value);  
}  
return root;  
}  
// Update the height & Balance Factor of each node
```

```
root->height =max[height(root->left), height(root->right)]+1;  
Balance = Balance Factor(root);  
// ROTATIONS  
if(balance>1 && value<root->left->value)// RR Rotation  
    return rightrotate(root);  
if(balance <-1&& value>root->right->value)// LL Rotation  
    return leftrotate(root);  
if(balance >1&& value>root->left->value)// LR Rotation  
    root->left= leftrotate(root->left);  
    return rightrotate(root);  
if(balance <-1&& value<root->right->value)// RL Rotation  
    root->right= rightrotate(root->right);  
    return leftrotate(root);  
return root;  
}
```



## School of Computer Science and Engineering

Winter Semester 2022-2023

### Continuous Assessment Test – 1 KEY

**SLOT A1**

**Programme Name &Branch: B. Tech ( BCI, BCE, BCT, BDS, BCB)**

**Course Name & code: Data Structures and Algorithms – BCSE202L**

**Class Number (s): VL2022230505842, 6331, 5847, 5851, 5837, 6304, 5855, 5849, 5840**

**Faculty Name (s): Joshva Devadas T, Priyanka N, Sayan Sikdar, Naveenkumar J, Akash Sinha, Kalaivani K, Ganesh Shamrao Khekare, Sunil Kumar PV, S M Farooq.**

**Exam Duration: 90 Min.**

**Maximum Marks: 50**

**All questions carry equal marks.**

<b>Q.No.</b>	<b>Question</b>	<b>Max Marks</b>
1.	<p>a) Find the time complexity for the given recurrence relation using Master's Theorem.</p> <p>i) <math>T(n) = 3T\left(\frac{n}{2}\right) + n^2</math>   ii) <math>T(n) = 2T\left(\frac{n}{2}\right) + n \log n</math></p> <p>Ans: Writing master's theorem</p> <ul style="list-style-type: none"><li>• It is used to analyse recurrence relation.</li><li>• <math>T(n) = aT\left(\frac{n}{b}\right) + \theta(n^k \log^p n)</math>, <math>a \geq 1, b &gt; 1, k \geq 0</math> and <math>p</math> is real number</li><li>• Case 1: if <math>a &gt; b^k</math>, then <math>T(n) = \theta(n^{\log_b a})</math></li><li>• Case 2: if <math>a = b^k</math>, then<ul style="list-style-type: none"><li>a) if <math>p &gt; -1</math>, then <math>T(n) = \theta(n^{\log_b a} \log^{p+1} n)</math></li><li>b) if <math>p = -1</math>, then <math>T(n) = \theta(n^{\log_b a} \log^2 n)</math></li><li>c) if <math>p &lt; -1</math>, then <math>T(n) = \theta(n^{\log_b a})</math></li></ul></li><li>Case 3: if <math>a &lt; b^k</math>, then<ul style="list-style-type: none"><li>a) if <math>p \geq 0</math>, then <math>T(n) = \theta(n^k \log^p n)</math></li><li>b) if <math>p &lt; 0</math>, then <math>T(n) = \theta(n^k)</math></li></ul></li></ul> <p>i)a=3, b=2, k=2, p=0 ( <math>a \geq 1, b &gt; 1, k \geq 0, p</math> is a real number)</p> $\begin{array}{ll} a & b^k \\ 3 & 2^2 \\ 3 & < 4 \Rightarrow a < b^k & \text{Falls under case 3a.} \end{array}$ $\begin{aligned} T(n) &= \theta(n^k \log^p n) \\ &= \theta(n^2 \log^0 n) \\ &= \theta(n^2) \end{aligned}$ <p>ii) a=2, b=2, k=1, p=1</p> $\begin{array}{ll} a & b^k \\ 2 & 2^1 \\ 2 & = 2 \Rightarrow a = b^k \end{array}$ <p>Falls under case 2a.</p> $\begin{aligned} T(n) &= \theta(n^{\log_b a} \log^{p+1} n) \\ &= \theta(n^{\log_2 2} \log^{1+1} n) \\ &= \theta(n^1 \log^2 n) = \theta(n \log^2 n) = \theta(n \log \log n) \end{aligned}$	<p style="color: red;">2 M</p> <p style="color: red;">2 M</p> <p style="color: red;">2 M</p>

b) Find time complexity for the following recursive relation using back substitution method. [For this question consider the following question for 4 M]

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1, & \text{for } n > 1 \\ 1, & \text{for } n = 1 \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Substitute  $n/2$  in place of  $n$  in the given recurrence relation

$$\begin{aligned} &= [T\left(\frac{\frac{n}{2}}{2}\right) + 1] + 1 \\ &= T\left(\frac{n}{4}\right) + 2 \\ &= T\left(\frac{n}{2^2}\right) + 2 \end{aligned}$$

Substitute  $n/4$  in place of  $n$  in the given recurrence relation

$$\begin{aligned} &= \left[ T\left(\frac{n}{8}\right) + 1 \right] + 2 \\ &= T\left(\frac{n}{8}\right) + 3 \\ &= T\left(\frac{n}{2^3}\right) + 3 \\ &\quad \dots \\ &= T\left(\frac{n}{2^k}\right) + k, [\text{ } k^{\text{th}} \text{ term }] \end{aligned}$$

When  $\frac{n}{2^k}$  reaches to 1 then it will be a base case.

$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$$

Substitute  $k$  value in  $k^{\text{th}}$  term.

$$\begin{aligned} &= T(1) + \log_2 n \\ &= 1 + \log_2 n \end{aligned}$$

Applying O notation to find time complexity,

$$= O(1 + \log_2 n) = O(\log_2 n)$$

2.

a) Find the time complexity for the following code (step count method)

i) sum1=0 for( k=1; k<=n;k=k*2) for(j=1; j<=n; j++) sum1++	ii) sum=0; for( j=1; j<=n; j++) for( i=1; i<=j; i++) sum++; for( k=0; k<n; k++) a[k]= k;
--	--

Ans:

i)

Code	Count
sum1=0	1
for( k=1; k<=n;k=k*2)	$\log n$
for(j=1; j<=n; j++)	n
sum1++	n
Total	$1 + (\log n) * (n+n)$ $1+ 2 n * \log n$
Applying Big O	$O(n \log n)$

4 M

2 M

ii)

For each j value i value is repeating for j times.

```
for( j=1; j<=n; j++)
    for( i=1; i<=j; i++)
        sum++;
```

3 M

i	1	2	3	4	--	n
J	1	2 times	3 times	4 times	--	n times
sum++	1*1	1*2	1*3	1*4	--	1*n times

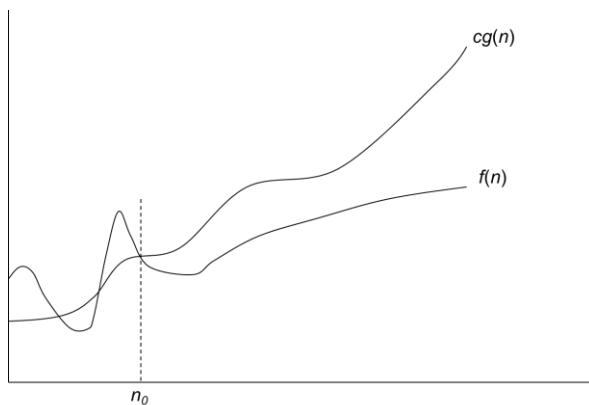
Finally,  $1 + 2 + 3 + 4 + 5 + \dots + n = n(n+1)/2$

Given Code	Count
sum=0; for( j=1; j<=n; j++) for( i=1; i<=j; i++) sum++; for( k=0; k<n; k++) a[k]= k;	1 $n(n+1)/2$ n n
Total	$1 + \frac{n*(n+1)}{2} + n + n$ $1 + \frac{n^2}{2} + \frac{n}{2} + 2n$
Applying Big O	$O(1 + \frac{n^2}{2} + \frac{n}{2} + 2n) = O(n^2)$

b) Find c and  $n_0$  values if  $f(n) = O(g(n))$  for the functions  $f(n) = 3n + 2$  and  $g(n) = n$ .

O notation:

2 M



$f(n) = O(g(n))$ : there exist positive constants c and  $n_0$  such that

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0$$

If  $f(n) = O(g(n))$ , then  $f(n) \leq c * g(n)$ , for  $c > 0$ ,  $n_0 \geq 1$

3 M

$$3n+2 \leq c*n$$

For  $c=4$ , the above equation holds.

$$3n+2 \leq 4n \rightarrow 2 \leq n \rightarrow n \geq 2.$$

3.	<p>Convert the given Infix notation into its equivalent postfix notation and show step trace with the conversion algorithm.  Infix notation: <math>(a/(b-c+d))^{*(e-a)}*c</math>  Ans:  Algorithm for conversion of Infix to postfix notation.</p> <p>Let, <b>X</b> is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression <b>Y</b>.</p> <ol style="list-style-type: none"> <li>1. Print operands as they arrive.</li> <li>2. If the stack is empty or contains a left parenthesis on top, push the Incoming operator onto the stack.</li> <li>3. If the Incoming symbol is '(', push it onto stack.</li> <li>4. If the Incoming symbol is ')', pop the stack &amp; add operators to output Y until left parenthesis is found.</li> <li>5. If incoming symbol has higher precedence than the top of the stack, push it onto stack.</li> <li>6. If the Incoming symbol has lower precedence than the top of the stack, then pop and add to the output. Then test the incoming operator against the new top of the stack.</li> <li>7. If the incoming operator has equal precedence with the top of the stack, use associativity rule.</li> <li>8. At the end of the expression pop all the operators of stack.</li> </ol> <p>Associativity <b>L to R</b> then pop &amp; print the top of the stack &amp; then push in the incoming operator.  <b>R to L</b> then push the incoming operator</p> <p style="text-align: center;"><b>Input Infix notation: <math>(a/(b-c+d))^{*(e-a)}*c</math></b></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">S. No.</th><th style="text-align: center;">Input Symbol</th><th style="text-align: center;">Rule Number</th><th style="text-align: center;">Stack</th><th style="text-align: center;">Output</th></tr> </thead> <tbody> <tr> <td style="text-align: center;">1</td><td style="text-align: center;">(</td><td style="text-align: center;">3</td><td style="text-align: center;">(</td><td style="text-align: center;">Empty</td></tr> <tr> <td style="text-align: center;">2</td><td style="text-align: center;">a</td><td style="text-align: center;">1</td><td style="text-align: center;">(</td><td style="text-align: center;">a</td></tr> <tr> <td style="text-align: center;">3</td><td style="text-align: center;">/</td><td style="text-align: center;">2</td><td style="text-align: center;">(/</td><td style="text-align: center;">a</td></tr> <tr> <td style="text-align: center;">4</td><td style="text-align: center;">(</td><td style="text-align: center;">3</td><td style="text-align: center;">(/()</td><td style="text-align: center;">a</td></tr> <tr> <td style="text-align: center;">5</td><td style="text-align: center;">b</td><td style="text-align: center;">1</td><td style="text-align: center;">(/()</td><td style="text-align: center;">a b</td></tr> <tr> <td style="text-align: center;">6</td><td style="text-align: center;">-</td><td style="text-align: center;">2</td><td style="text-align: center;">(/() -</td><td style="text-align: center;">a b</td></tr> <tr> <td style="text-align: center;">7</td><td style="text-align: center;">c</td><td style="text-align: center;">1</td><td style="text-align: center;">(/() -</td><td style="text-align: center;">a b c</td></tr> <tr> <td style="text-align: center;">8</td><td style="text-align: center;">+</td><td style="text-align: center;">7 &amp; Associativity rule</td><td style="text-align: center;">(/() +</td><td style="text-align: center;">a b c -</td></tr> <tr> <td style="text-align: center;">9</td><td style="text-align: center;">d</td><td style="text-align: center;">1</td><td style="text-align: center;">(/() +</td><td style="text-align: center;">a b c - d</td></tr> <tr> <td style="text-align: center;">10</td><td style="text-align: center;">)</td><td style="text-align: center;">4</td><td style="text-align: center;">(/</td><td style="text-align: center;">a b c - d +</td></tr> <tr> <td style="text-align: center;">11</td><td style="text-align: center;">)</td><td style="text-align: center;">4</td><td style="text-align: center;">Empty</td><td style="text-align: center;">a b c - d + /</td></tr> <tr> <td style="text-align: center;">12</td><td style="text-align: center;">*</td><td style="text-align: center;">5</td><td style="text-align: center;">*</td><td style="text-align: center;">a b c - d + /</td></tr> <tr> <td style="text-align: center;">13</td><td style="text-align: center;">(</td><td style="text-align: center;">3</td><td style="text-align: center;">* (</td><td style="text-align: center;">a b c - d + /</td></tr> <tr> <td style="text-align: center;">14</td><td style="text-align: center;">e</td><td style="text-align: center;">1</td><td style="text-align: center;">* (</td><td style="text-align: center;">a b c - d + / e</td></tr> <tr> <td style="text-align: center;">15</td><td style="text-align: center;">-</td><td style="text-align: center;">2</td><td style="text-align: center;">* ( -</td><td style="text-align: center;">a b c - d + / e</td></tr> <tr> <td style="text-align: center;">16</td><td style="text-align: center;">a</td><td style="text-align: center;">1</td><td style="text-align: center;">* ( -</td><td style="text-align: center;">a b c - d + / e a</td></tr> <tr> <td style="text-align: center;">17</td><td style="text-align: center;">)</td><td style="text-align: center;">4</td><td style="text-align: center;">*</td><td style="text-align: center;">a b c - d + / e a -</td></tr> <tr> <td style="text-align: center;">18</td><td style="text-align: center;">*</td><td style="text-align: center;">7 &amp; Associativity rule</td><td style="text-align: center;">*</td><td style="text-align: center;">a b c - d + / e a - *</td></tr> <tr> <td style="text-align: center;">19</td><td style="text-align: center;">c</td><td style="text-align: center;">1</td><td style="text-align: center;">*</td><td style="text-align: center;">a b c - d + / e a - * c</td></tr> <tr> <td style="text-align: center;">20</td><td></td><td style="text-align: center;">8</td><td style="text-align: center;">Empty</td><td style="text-align: center;">a b c - d + / e a - * c *</td></tr> </tbody> </table>	S. No.	Input Symbol	Rule Number	Stack	Output	1	(	3	(	Empty	2	a	1	(	a	3	/	2	(/	a	4	(	3	(/()	a	5	b	1	(/()	a b	6	-	2	(/() -	a b	7	c	1	(/() -	a b c	8	+	7 & Associativity rule	(/() +	a b c -	9	d	1	(/() +	a b c - d	10	)	4	(/	a b c - d +	11	)	4	Empty	a b c - d + /	12	*	5	*	a b c - d + /	13	(	3	* (	a b c - d + /	14	e	1	* (	a b c - d + / e	15	-	2	* ( -	a b c - d + / e	16	a	1	* ( -	a b c - d + / e a	17	)	4	*	a b c - d + / e a -	18	*	7 & Associativity rule	*	a b c - d + / e a - *	19	c	1	*	a b c - d + / e a - * c	20		8	Empty	a b c - d + / e a - * c *	3 M  7 M
S. No.	Input Symbol	Rule Number	Stack	Output																																																																																																							
1	(	3	(	Empty																																																																																																							
2	a	1	(	a																																																																																																							
3	/	2	(/	a																																																																																																							
4	(	3	(/()	a																																																																																																							
5	b	1	(/()	a b																																																																																																							
6	-	2	(/() -	a b																																																																																																							
7	c	1	(/() -	a b c																																																																																																							
8	+	7 & Associativity rule	(/() +	a b c -																																																																																																							
9	d	1	(/() +	a b c - d																																																																																																							
10	)	4	(/	a b c - d +																																																																																																							
11	)	4	Empty	a b c - d + /																																																																																																							
12	*	5	*	a b c - d + /																																																																																																							
13	(	3	* (	a b c - d + /																																																																																																							
14	e	1	* (	a b c - d + / e																																																																																																							
15	-	2	* ( -	a b c - d + / e																																																																																																							
16	a	1	* ( -	a b c - d + / e a																																																																																																							
17	)	4	*	a b c - d + / e a -																																																																																																							
18	*	7 & Associativity rule	*	a b c - d + / e a - *																																																																																																							
19	c	1	*	a b c - d + / e a - * c																																																																																																							
20		8	Empty	a b c - d + / e a - * c *																																																																																																							

<p>4.</p> <p>a) List out the advantages of circular queue. (Any three enough)</p> <p>Ans: Page replacement algorithms CPU Scheduling Inter-process communication Resource allocation in operating systems</p> <p>b) Engrave pseudocode to perform operations on circular Queue. Assume the initial state of the Circular Queue is Empty with size N=4 and demonstrate the following operations with an appropriate diagram with the status of REAR and FRONT pointers.</p> <p>Instructions in sequence:</p> <ol style="list-style-type: none"> <li>1. Successive Insertion of the elements 10, 20, 30, 40, 50.</li> <li>2. Successive Deletion of two elements.</li> <li>3. Insertion of the element 50.</li> <li>4. Successive deletion of three elements.</li> </ol> <p>Ans:</p> <p>Let us assume an array CQ of size MAX with initial FRONT values REAR are -1.</p> <p>Algorithm CQ_Insert(ITEM)</p> <p>Step 1: IF (REAR+1)%MAX = FRONT THEN //If REAR reaches to FRONT     Write “circular queue overflow”     Exit     [END OF IF]</p> <p>Step 2: IF REAR = -1 and FRONT = -1 THEN // If Circular Queue is Empty.     REAR = FRONT=0     ELSE         REAR = (REAR+1)%MAX // Increment REAR value for Insert     [END OF IF]</p> <p>Step 3: CQ[REAR]=ITEM //Insertion of ITEM into CQ</p> <p>Step 4: Exit</p> <p>Algorithm CQ_Delete( )</p> <p>Step 1: IF FRONT=-1 and REAR=-1 THEN // If CQ is empty     Write “circular queue under flow”     Exit     [END OF IF]</p> <p>Step 2: SET VAL = CQ[FRONT] // Deleted element in VAL</p> <p>Step 3: IF FRONT = REAR THEN // If CQ has a single element     REAR = FRONT = -1     ELSE         FRONT = (FRONT+1)%MAX     [END OF IF]</p> <p>Step 4: Exit</p> <ol style="list-style-type: none"> <li>1. Successive Insertion of 10, 20, 30, 40, 50</li> </ol>	<p><b>3 M</b></p> <p><b>3 M</b></p>
---	-------------------------------------

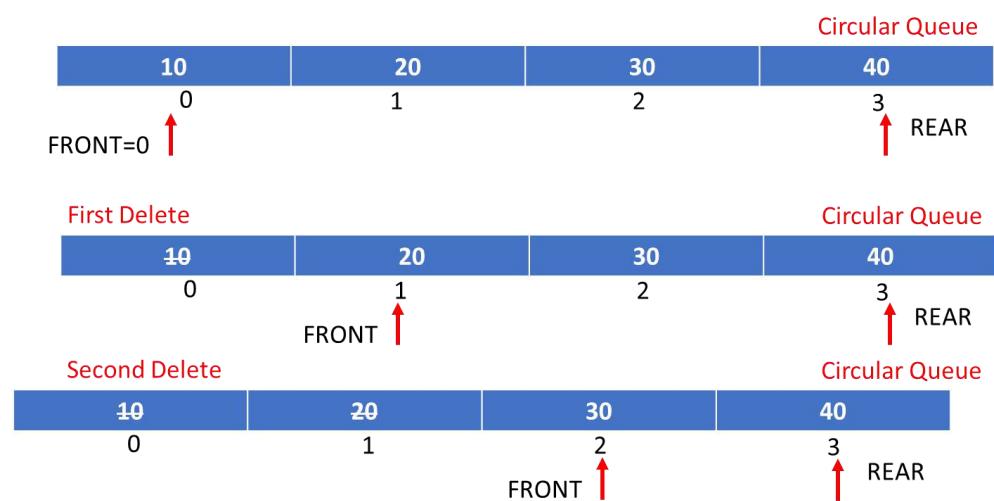
Successive Insertion of 10, 20, 30, 40, 50  
 REAR = FRONT = -1, MAX=4



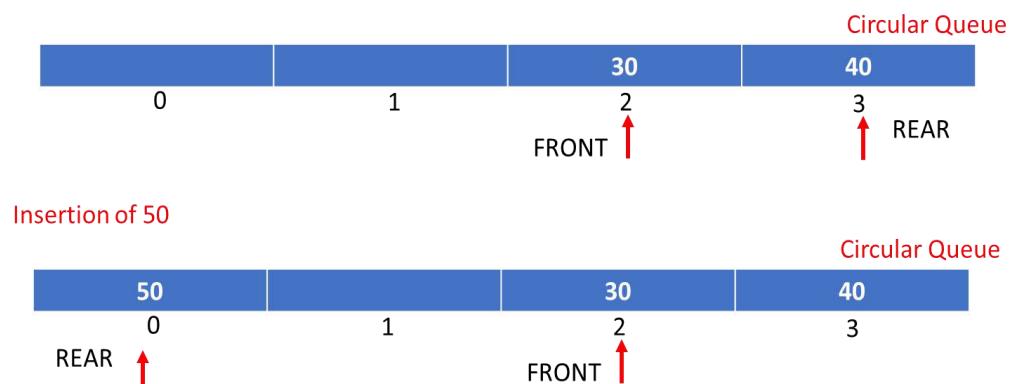
4 M

## 2. Successive deletion of two elements

Successive Deletion of two elements

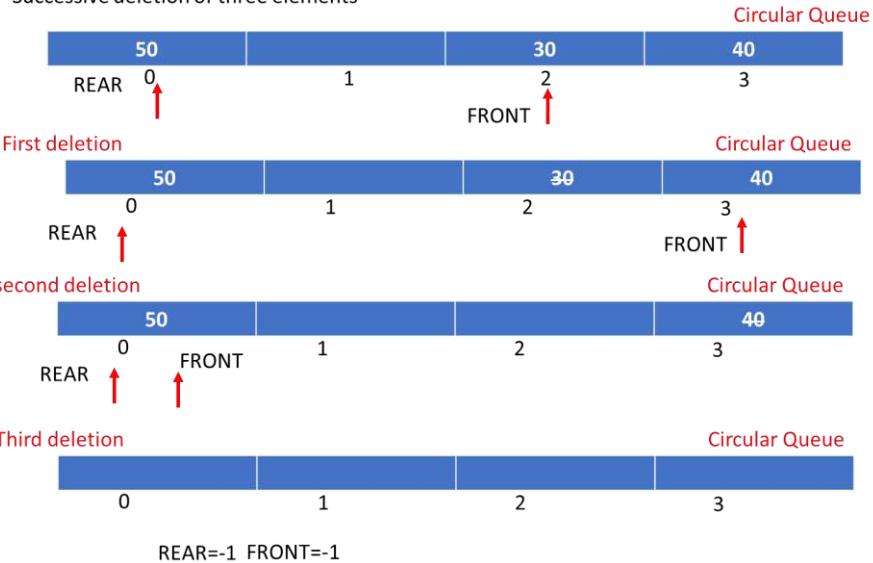


## 3. Insertion of the element 50.



4. Successive deletion of three elements

Successive deletion of three elements



REAR=-1 FRONT=-1

5. Analyze Insertion sort algorithm. Sort and give the trace for the following elements using Insertion sort.

C, A, Z, S, P, S  
67, 65, 90, 83, 80, 83 (ASCII Values)

*Insertion\_Sort(A,N)*

*step 1: Repeat step 2 to 5 for k: = 1 to N-1*

*step 2: set TEMP:= A[k]*

*step 3: set J:= K - 1*

*step 4: Repeat while TEMP <= A[J]*

*set A[J+1]:= A[J]*

*set J:= J - 1*

*[End of Inner While loop]*

*step 5: set A[J+1] = TEMP*

3 M

Let us an Array A of N elements is in memory. Insertion sort is one of the simplest sorting algorithms. It consists of N-1 passes.

Original	67	65	90	83	80	83	Positions moved
pass = 1	65	67	90	83	80	83	1
Pass = 2	65	67	90	83	80	83	0

	Pass = 3	<b>65</b>	<b>67</b>	<b>83</b>	<b>90</b>	<b>80</b>	<b>83</b>	1	4 M
	Pass = 4	65	67	80	83	90	83	2	
	Pass = 5	65	67	80	83	83	90	2	

Sorted elements: A, C, P, S, S, Z  
 Time complexity: For each iteration (pass) no. of comparisons are  
 1 , 2, 3, 4, ----n-1  
 Adding those comparisons ( 1+2+3+ --- n-1) =  $n(n-1)/2= O(n^2)$   
 Best case time complexity:  $\Omega(n)$  (If all the elements are presorted)  
 Average case time complexity  $\theta(n^2)$ .

\*\*End of key\*\*



**School of Computer**

**Science and**

**Engineering**

**Winter Semester 2022-2023**  
**Continuous Assessment Test – 2**

**SLOT: A1**

**Programme Name & Branch:** B.Tech – CSE ( BCI, BCE, BCT, BDS, BCB)

**Course Name & code:** Data Structures and Algorithms – BCSE202L

**Class Number (s):** VL2022230505842, 6331, 5847, 5851, 5837, 6304, 5855, 5849, 5840

**Faculty Name (s):** Joshva Devadas T, Priyanka N, Sayan Sikdar, Naveenkumar J, Akash Sinha, Kalaivani K, Ganesh Shamrao Khekare, Sunil Kumar PV, S M Farooq.

**Exam Duration: 90 Min.**

**Maximum Marks: 50**

**General instruction(s):**

All Questions carry Equal Marks.

Q.No.	Question	Max Marks
1.	<p>a) You are given a Singly Linked list below.  <math>\text{Head} \rightarrow A \rightarrow B \rightarrow A \rightarrow B \rightarrow A \rightarrow B \rightarrow \text{NULL}</math></p> <p>Write the procedure to perform an alternating split of the given singly linked list. The expected output after splitting the singly linked list should result in two singly linked lists each containing only similar characters.</p> <p style="text-align: right;">(4 Marks)</p> <p>b) Write a procedure to check if two circular linked lists are identical. What will be the time complexity of this procedure. Illustrate the working of the procedure using the given circular linked list.</p> <p style="text-align: center;">List 1: 1 -&gt; 2 -&gt; 3 -&gt; 1</p> <p style="text-align: center;">List 2: 1 -&gt; 2 -&gt; 3 -&gt; 1</p> <p style="text-align: right;">(6 Marks)</p>	10
Sol.	<p>a) <b>Procedure: 3 Marks</b></p> <ol style="list-style-type: none"> <li>1. Initialize two new linked lists, listA and listB, to store the resulting split lists.</li> <li>2. Initialize a pointer current to the head of the input linked list.</li> <li>3. While current is not NULL:           <ol style="list-style-type: none"> <li>a. Append the current node to listA.</li> <li>b. Move current to its next node.</li> <li>c. If current is not NULL:               <ol style="list-style-type: none"> <li>i. Append the current node to listB.</li> <li>ii. Move current to its next node.</li> </ol> </li> </ol> </li> <li>4. Set the next pointers of the last nodes in listA and listB to NULL.</li> </ol> <p style="color: red;"><b>After performing this procedure on the given singly linked list, we get two</b></p>	

**new singly linked lists : 1 Mark**

listA: Head -> A -> A -> A -> NULL

listB: Head -> B -> B -> B -> NULL

**b) Procedure: 3 Marks**

1. Initialize two pointers, one for each list. Let's call them ptr1 and ptr2. Set ptr1 to point to the first node (START1) of List 1 and ptr2 to point to the first node (START2) of List 2.
2. While ptr1 is not equal to the first node of List 1 or ptr2 is not equal to the first node of List 2:
  - a. Compare the data of the current nodes pointed to by ptr1 and ptr2.
  - b. If the data is not the same, return false.
  - c. Move both pointers PTR1 and PTR2 to their next nodes.
3. If both pointers reach their starting points (START1 and START2) at the same time, return true. Otherwise, return false.

**Complexity: 1 Mark**

The time complexity of this procedure is  $O(n)$  where n is the number of nodes in the larger list.

**Illustration: 2 Marks**

- In this illustration, the data of the first nodes pointed to by ptr1 and ptr2 is the same (both are 1),
  - so move both pointers to their next nodes.
- In the next iteration of the loop, the data of the current nodes pointed to by ptr1 and ptr2 is again the same (both are 2),
  - so move both pointers to their next nodes.
- In the third iteration of the loop, the data of the current nodes pointed to by ptr1 and ptr2 is again the same (both are 3),
  - so move both pointers to their next nodes.
- At this point, both pointers have reached their starting points (the first node of their respective lists), so exit the loop and return true since both lists are identical.

2.	Suppose you are given an array of strings that need to be sorted in lexicographic order (i.e., dictionary order). How can you modify the quicksort algorithm to handle this requirement? Illustrate the working of your modified quicksort algorithm on the following array of names. names = ["Smith", "Johnson", "Williams", "Brown", "Jones", "Garcia", "Miller", "Davis"]	10
Sol.	<b>Modified Quick Sort Algorithm Idea: 2 Marks</b> Quicksort can be easily modified to sort an array of strings in lexicographic order. The only change required is in the comparison function used to compare two elements of the array. Instead of using the	

standard comparison operators (e.g., `<`, `>`), need to use a string comparison function such as `strcmp` in C

#### Modified Quick Sort Algorithm/Pseudocode: 5 Marks

```
// Function to swap two strings in an array
void swap(char* arr[], int i, int j) {
    char* temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

// Partition function for quicksort
int partition(char* arr[], int low, int high) {
    char* pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (strcmp(arr[j], pivot) < 0) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i+1, high);
    return i+1;
}

// Modified quicksort function for lexicographic sorting of an array of
// strings
void quicksort_lex(char* arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort_lex(arr, low, pi-1);
        quicksort_lex(arr, pi+1, high);
    }
}
```

#### Illustration Steps: 3 Marks

- Choose a pivot element, say “Johnson”.
- Partition the array into two sub-arrays: elements that are lexicographically less than “Johnson” and elements that are lexicographically greater than “Johnson”. After partitioning, the array becomes [“Garcia”, “Davis”, “Brown”, “Johnson”, “Jones”, “Miller”, “Smith”, “Williams”].
- Recursively apply the modified quicksort algorithm to the two sub-arrays.

```
char* arr[] = {"Smith", "Johnson", "Williams", "Brown", "Jones",
"Garcia", "Miller", "Davis"};
int n = sizeof(arr) / sizeof(arr[0]);
```

	quicksort_lex(arr, 0, n-1);	
3.	<p><b>a)</b> Construct the Binary tree from given Traversals. (4 Marks)</p> <p>Inorder Traversal : { 4, 2, 1, 7, 5, 8, 3, 6 }</p> <p>Postorder Traversal: { 4, 2, 7, 8, 5, 6, 3, 1 }</p> <p><b>b)</b> Build a expression tree from the given expression: (6 Marks)</p> <p><math>((5 * (y^2)) - (3 * y)) + 2,</math></p> <p>Also using the built expression tree extract the Postfix and prefix Expressions.</p>	10
Sol.	<p><b>a) Binary tree – 4 Marks.</b></p> <pre> graph TD     1((1)) --&gt; 2((2))     1 --&gt; 3((3))     2 --&gt; 4((4))     3 --&gt; 5((5))     3 --&gt; 6((6))     5 --&gt; 7((7))     5 --&gt; 8((8))   </pre> <p><b>b) Expression Tree – 4 Marks.</b></p> <pre> graph TD     plus["+"] --- slash1["/"]     plus --- backslash1["\""]     slash1 --- minus["-"]     slash1 --- two["2"]     minus --- star1["*"]     minus --- backslash2["\""]     star1 --- five["5"]     star1 --- caret["^"]     caret --- y1["y"]     caret --- two["2"]     backslash2 --- star2["*"]     backslash2 --- y2["y"]     star2 --- three["3"]     star2 --- y2   </pre> <p>Postfix - 5 y 2 ^ * 3 y * - 2 + 1 Marks    Prefix - + - * 5 ^ y 2 * 3 y 2 1 Marks</p>	
4.	<p>a) Suppose you have a binary search tree containing the ages of all the employees in a company. The tree is balanced and has n nodes. You need to write an algorithm/pseudocode to find the 3rd youngest employee in the company. Illustrate your algorithm's working using the following binary search tree containing the ages of the employees.</p>	10

	<p>What is the time complexity of your algorithm for finding the 3rd youngest employee in the company.</p> <pre>         40         /   \       30     50       / \   / \     20  35  45  55 </pre> <p>(5 Marks)</p> <p>b) Create an algorithm and write pseudocode to determine if a given binary tree is also a binary search tree. The algorithm should have a time complexity of <math>O(n)</math> and a space complexity of <math>O(n)</math>. (5 Marks)</p>	
Sol.	<p>a) algorithm to find the kth smallest element in a binary search tree.</p> <p><b>Algorithm or Pseudocode – 2 Marks</b></p> <ul style="list-style-type: none"> <li>• Perform an inorder traversal of the binary search tree. This will visit the nodes of the tree in ascending order of their values.</li> <li>• Keep track of a counter variable as you traverse the tree. Increment the counter each time you visit a node.</li> <li>• When the counter reaches k, stop the traversal, and return the value of the current node.</li> </ul> <pre> function kthSmallest(root, k)   stack = empty stack   node = root   while (node is not null or stack is not empty)     while (node is not null)       push node onto stack       node = node.left     node = pop stack     k = k - 1     if k == 0       return node.value     node = node.right </pre> <p><b>Illustration – 2 Marks</b></p> <ul style="list-style-type: none"> <li>• Let's say we want to find the 3rd youngest employee in the company (i.e., <math>k = 3</math>).</li> <li>• start by performing an inorder traversal of the binary search tree. This visits the nodes in ascending order: 20, 30, 35, 40, 45, 50, 55.</li> <li>• keep track of a counter variable as traversing the tree. The counter starts at 0 and is incremented each time we visit a node.</li> <li>• When we visit the first node (20), the counter becomes 1. When we visit the second node (30), the counter becomes 2. When we visit the third node (35), the counter becomes 3.</li> </ul>	

- Since  $k = 3$ , stop the traversal at this point and return the value of the current node (35).
- Therefore, the age of the 3rd youngest employee in the company is 35.

### Time Complexity – 1 Mark

The time complexity of the algorithm to find the k-th smallest element in a binary search tree (BST) is  $O(h + k)$ , where  $h$  is the height of the BST and  $k$  is the rank of the element that is to be found.

The algorithm performs an in-order traversal of the BST using a stack.

Each node is visited once during the traversal, and the time it takes to visit a node is  $O(1)$ . So, the total time it takes to visit all nodes is  $O(n)$ , where  $n$  is the number of nodes in the BST.

However, we don't need to visit all nodes in the BST. Required to only visit the first  $k$  nodes in the in-order traversal. So, the time complexity of the algorithm is  $O(h + k)$ , where  $h$  is the height of the BST and  $k$  is the rank of the element needed to be found.

b) A better solution is to look at each node only once. The trick is to write a utility helper function `IsBSTUtil(struct BinaryTreeNode* root, int min, int max)` that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be `INT_MIN` and `INT_MAX` – they narrow from there. (2 Marks)

### Pseudocode: 3 Marks

```

Initial call: IsBST(root, INT_MIN, INT_MAX);
int IsBST(struct BinaryTreeNode *root, int min, int max) {
    if(!root)
        return 1;
    return (root->data > min && root->data < max &&
            IsBSTUtil(root->left, min, root->data) &&
            IsBSTUtil(root->right, root->data, max));
}

```

### OR

By using inorder traversal. The idea behind this solution is that inorder traversal of BST produces sorted lists. While traversing the BST in inorder, at each node check the condition that its key value should be greater than the key value of its previous visited node. Also, we need to initialize the prev with possible minimum integer value (say, `INT_MIN`). (2 Marks)

### Pseudocode: 3 Marks

	<pre> int prev = INT_MIN; int IsBST(struct BinaryTreeNode *root, int *prev) {     if(!root) return 1;     if(!IsBST(root-&gt;left, prev))         return 0;     if(root-&gt;data &lt; *prev)         return 0;     *prev = root-&gt;data;     return IsBST(root-&gt;right, prev); } </pre>	
5.	<p>Assume you are given an array of integers representing the ages of people in a group: [25, 32, 40, 19, 60, 55, 22]. You need to sort this array in ascending order using heap sort with a min heap. Additionally, after sorting the array, two new people join the group with ages 35 and 18. You need to insert these two new ages into the sorted array while maintaining the sorted order.</p> <p>i) How would you perform heap sort on the given array using a min heap? Illustrate and write the pseudocode.</p> <p>ii) How would you insert the two new ages into the sorted array while maintaining the sorted order? Illustrate and write the pseudocode.</p>	10
Sol.	<p>i) <b>Procedure: 1 Mark</b></p> <ol style="list-style-type: none"> <li>Build a min heap from the input array. It can be done by starting from the last non-leaf node and moving upwards, calling the min-heapify function on each node. The min-heapify function ensures that the subtree rooted at the current node is a min heap by recursively moving the smallest element to the root.</li> <li>Repeatedly extract the minimum element from the min heap and move it to the end of the array. It can be done by swapping the first element (the minimum element) with the last element in the heap and then reducing the size of the heap by one. After each extraction, call the min-heapify function on the root to restore the min heap property.</li> </ol> <p><b>Pseudocode: 2 Marks</b></p> <pre> void buildMinHeap(int arr[], int n) {     for (int i = n/2 - 1; i &gt;= 0; i--) {         minHeapify(arr, n, i);     } }  void minHeapify(int arr[], int n, int i) {     int left = 2*i + 1;     int right = 2*i + 2;     int smallest = i;     if (left &lt; n &amp;&amp; arr[left] &lt; arr[smallest]) {         smallest = left;     }     if (right &lt; n &amp;&amp; arr[right] &lt; arr[smallest]) {         smallest = right;     }     if (smallest != i) {         swap(arr[i], arr[smallest]);         minHeapify(arr, n, smallest);     } } </pre>	

```

    }
    if (right < n && arr[right] < arr[smallest]) {
        smallest = right;
    }
    if (smallest != i) {
        swap(arr[i], arr[smallest]);
        minHeapify(arr, n, smallest);
    }
}

void heapsort(int arr[], int n) {
    buildMinHeap(arr, n);
    for (int i = n-1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        minHeapify(arr, i, 0);
    }
}

```

### Illustration: 3 Marks

Given elements are [25, 32, 40, 19, 60, 55, 22].

I1---- 25,32,40,19,60,55,22 => 25,32,22,19,60,55,40

I2---- 25,32,22,19,60,55,40 => 25,19,22,32,60,55,40

I3---- 25,19,22,32,60,55,40 => 19,25,22,32,60,55,40 (Min heap)

### Heap Sort

19,25,22,32,60,55,40 --- swap --- 40,25,22,32,60,55,19 extract 19

40,25,22,32,60,55 --- Heapify --- 22, 25, 40, 32, 60, 55

22, 25, 40, 32, 60, 55 --- swap --- 55, 25, 40, 32, 60, 22 extract 22

55, 25, 40, 32, 60 --- Heapify --- 25, 32, 40, 55, 60

25, 32, 40, 55, 60 --- swap --- 60, 32, 40, 55, 25 extract 25

60,32,40,55 --- Heapify --- 32, 55, 40, 60

32, 55, 40, 60 --- Swap --- 60, 55, 40, 32 extract 32

60, 55, 40 --- heapify --- 40, 55, 60

40, 55, 60 --- swap --- 60, 55, 40 extract 40

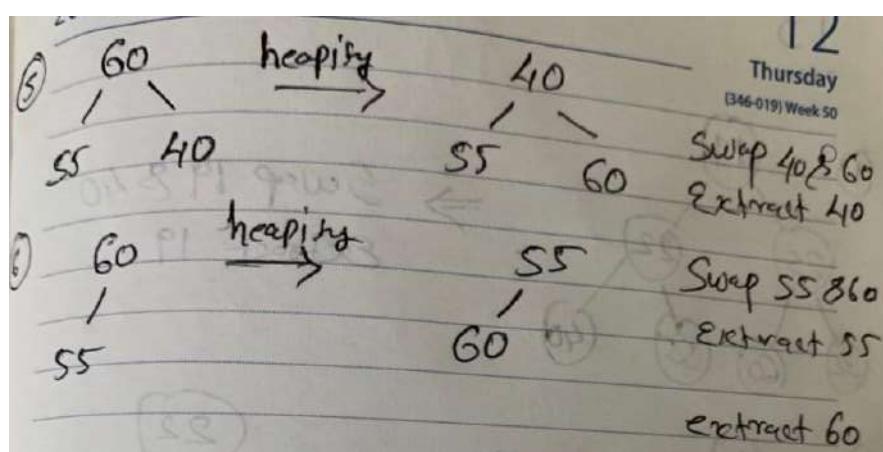
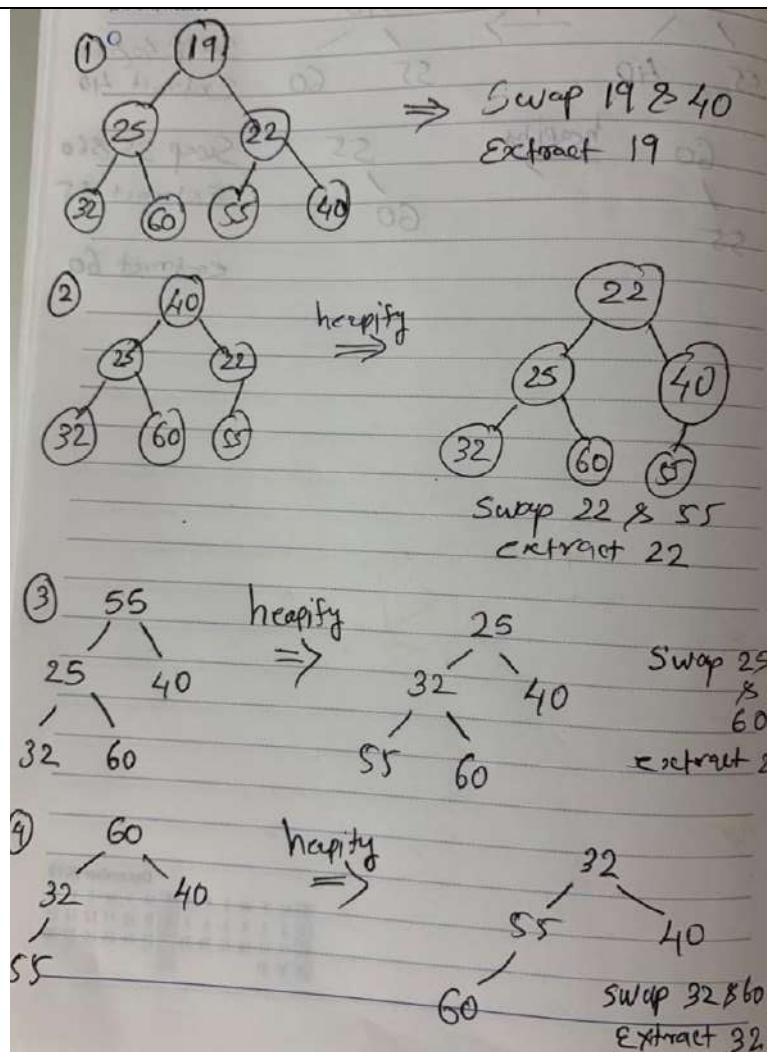
60, 55 --- heapify --- 55, 60

55, 60 --- swap --- 60, 55 extract 55

60 – extract 60

Sorted Array- 19, 22, 25, 32, 40, 55, 60

OR



## ii) Procedure: 2 Marks

To insert the two new ages into the sorted array while maintaining the sorted order, we can follow these steps:

- Append the new ages to the end of the array.
- Re-sort the array using the heap sort algorithm.

## Illustration: 2 Marks

Insert the new elements 35 and 18 into the min heap. The min heap would look like this after inserting 35:

```
 19
 / \
22  25
/\  /\
32 40 55 60
/
35
```

After inserting element 18:

```
 18
 / \
19  25
/\  /\
22 40 55 60
/\ 
32 35
```

Extract the minimum element from the min heap and insert it at the end of the sorted array until the heap is empty. The resulting sorted array would be: [18,19,22,25,32,35,40,55,60].

## **SINGLE LINKED LIST**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;

void begininsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void main ()
{
    int choice =0;
    while(choice != 9)
    {
        printf("\n\n*****Main Menu*****\n");
        printf("\nChoose one option from the following list ... \n");
        printf("=====\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n 5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
                begininsert();
                break;
            case 2:
```

```
lastinsert();
break;
case 3:
randominsert();
break;
case 4:
begin_delete();
break;
case 5:
last_delete();
break;
case 6:
random_delete();
break;

case 7:
display();
break;
case 8:
exit(0);
break;
default:
printf("Please enter valid choice..");

}

}

}

void begininsert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
{
```

```

printf("\nEnter value\n");
scanf("%d",&item);
ptr->data = item;
ptr->next = head;
head = ptr;
printf("\nNode inserted");

}

void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            ptr -> next = NULL;
            head = ptr;
            printf("\nNode inserted");
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
            {
                temp = temp -> next;
            }
            temp->next = ptr;
        }
    }
}

```

```

ptr->next = NULL;
printf("\nNode inserted");

}

}

}

void randominsert()
{
    int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }
        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}

```

```

}

void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from the begining ...\\n");
    }
}

void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ...\\n");
    }

    else
    {
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        free(ptr1);
    }
}

```

```

    }
    ptr1->next = NULL;
    free(ptr);
    printf("\nDeleted Node from the last ...\\n");
}
}

void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \\n")
    ;
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}

void display()
{
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
        printf("Nothing to print");
    }
}

```

```

    }
else
{
    printf("\nprinting values . . . .\n");
    while (ptr!=NULL)
    {
        printf("\n%d",ptr->data);
        ptr = ptr -> next;
    }
}
}
}

```

## **DOUBLE LINKED LIST**

```

#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node *prev;
    struct node *next;
    int data;
};

struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void main ()
{
int choice =0;
while(choice != 9)
{
    printf("\n*****Main Menu*****\n");
}
}

```

```
printf("\nChoose one option from the following list ...\\n");
printf("=====\\n");
printf("1.Insert in begining\\n2.Insert at last\\n3.Insert at any random location\\n4.D
elete from Beginning\\n
5.Delete from last\\n6.Delete the node after the given data\\n7.Search\\n8.Show\\n9.Ex
it\\n");
printf("\nEnter your choice?\\n");
scanf("\\%d",&choice);
switch(choice)
{
    case 1:
        insertion_beginning();
    break;
    case 2:
        insertion_last();
    break;
    case 3:
        insertion_specified();
    break;
    case 4:
        deletion_beginning();
    break;
    case 5:
        deletion_last();
    break;
    case 6:
        deletion_specified();
    break;
    case 7:
        display();
    break;
    case 8:
        exit(0);
    break;
    default:
        printf("Please enter valid choice..");
}
```

```

        }
    }
}

void insertion_beginning()
{
    struct node *ptr;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter Item value");
        scanf("%d",&item);

        if(head==NULL)
        {
            ptr->next = NULL;
            ptr->prev=NULL;
            ptr->data=item;
            head=ptr;
        }
        else
        {
            ptr->data=item;
            ptr->prev=NULL;
            ptr->next = head;
            head->prev=ptr;
            head=ptr;
        }
        printf("\nNode inserted\n");
    }
}

void insertion_last()

```

```

{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value");
        scanf("%d",&item);
        ptr->data=item;
        if(head == NULL)
        {
            ptr->next = NULL;
            ptr->prev = NULL;
            head = ptr;
        }
        else
        {
            temp = head;
            while(temp->next!=NULL)
            {
                temp = temp->next;
            }
            temp->next = ptr;
            ptr ->prev=temp;
            ptr->next = NULL;
        }
    }
    printf("\nnode inserted\n");
}
void insertion_specified()
{
    struct node *ptr,*temp;

```

```

int item,loc,i;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
    printf("\n OVERFLOW");
}
else
{
    temp=head;
    printf("Enter the location");
    scanf("%d",&loc);
    for(i=0;i<loc;i++)
    {
        temp = temp->next;
        if(temp == NULL)
        {
            printf("\n There are less than %d elements", loc);
            return;
        }
    }
    printf("Enter value");
    scanf("%d",&item);
    ptr->data = item;
    ptr->next = temp->next;
    ptr -> prev = temp;
    temp->next = ptr;
    temp->next->prev=ptr;
    printf("\nnode inserted\n");
}
void deletion_beginning()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
}

```

```

else if(head->next == NULL)
{
    head = NULL;
    free(head);
    printf("\nnode deleted\n");
}

else
{
    ptr = head;
    head = head -> next;
    head -> prev = NULL;
    free(ptr);
    printf("\nnode deleted\n");
}

void deletion_last()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
    }
}

```

```

        free(ptr);
        printf("\nnode deleted\n");
    }
}

void deletion_specified()
{
    struct node *ptr, *temp;
    int val;
    printf("\n Enter the data after which the node is to be deleted : ");
    scanf("%d", &val);
    ptr = head;
    while(ptr -> data != val)
        ptr = ptr -> next;
    if(ptr -> next == NULL)
    {
        printf("\nCan't delete\n");
    }
    else if(ptr -> next -> next == NULL)
    {
        ptr ->next = NULL;
    }
    else
    {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nnode deleted\n");
    }
}
void display()
{
    struct node *ptr;
    printf("\n printing values...\n");
    ptr = head;
    while(ptr != NULL)
    {

```

```
printf("%d\n",ptr->data);
ptr=ptr->next;
}
}
}
```



Vellore - 632 014, Tamil Nadu, India

**SCHOOL OF COMPUTING SCIENCE AND ENGINEERING**

**DIGITAL ASSIGNMENT– I, WINTER 2022-23**

**Programme : B.Tech**

**Max.Marks : 10**

**Course : Data Structures and Algorithms**

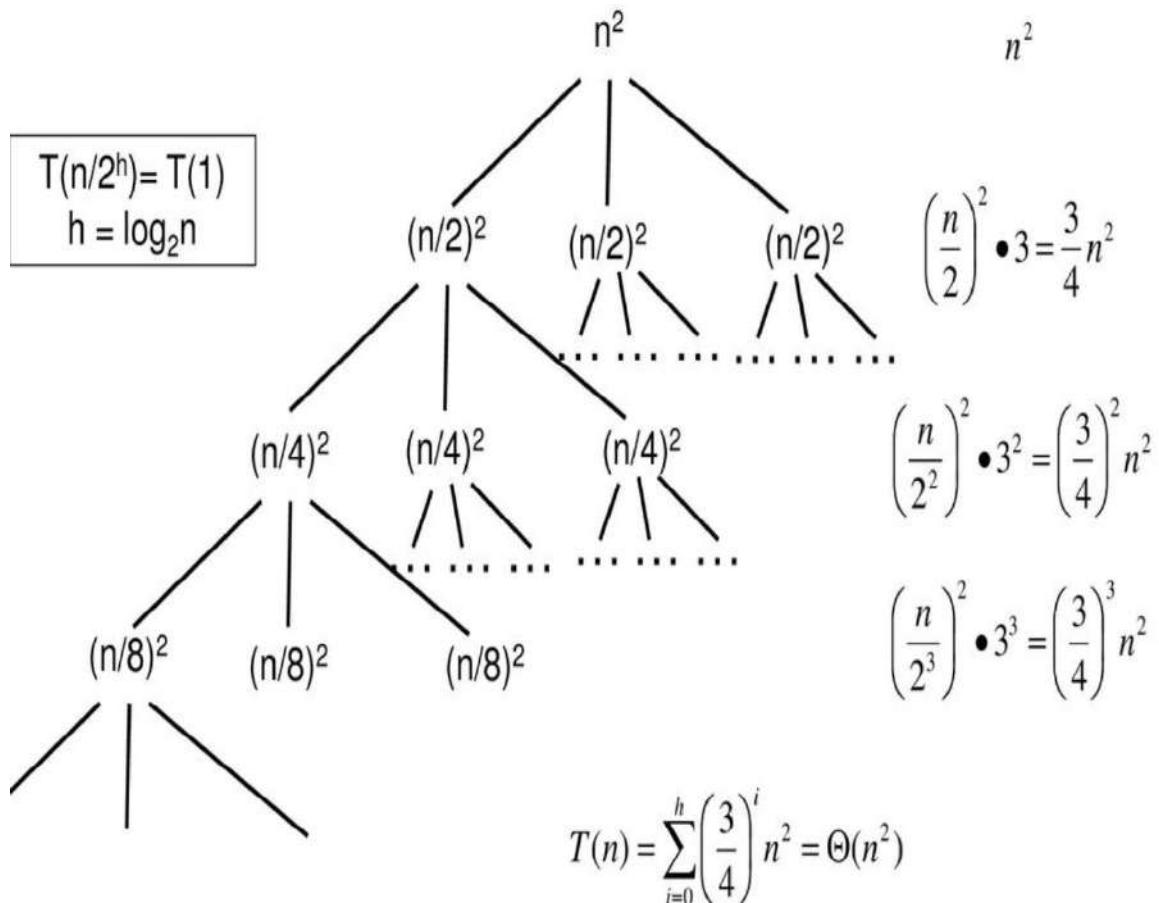
**Course Code : CSE202L**

**Questions**

1. Find the complexity of the below recurrence equation using the tree method

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

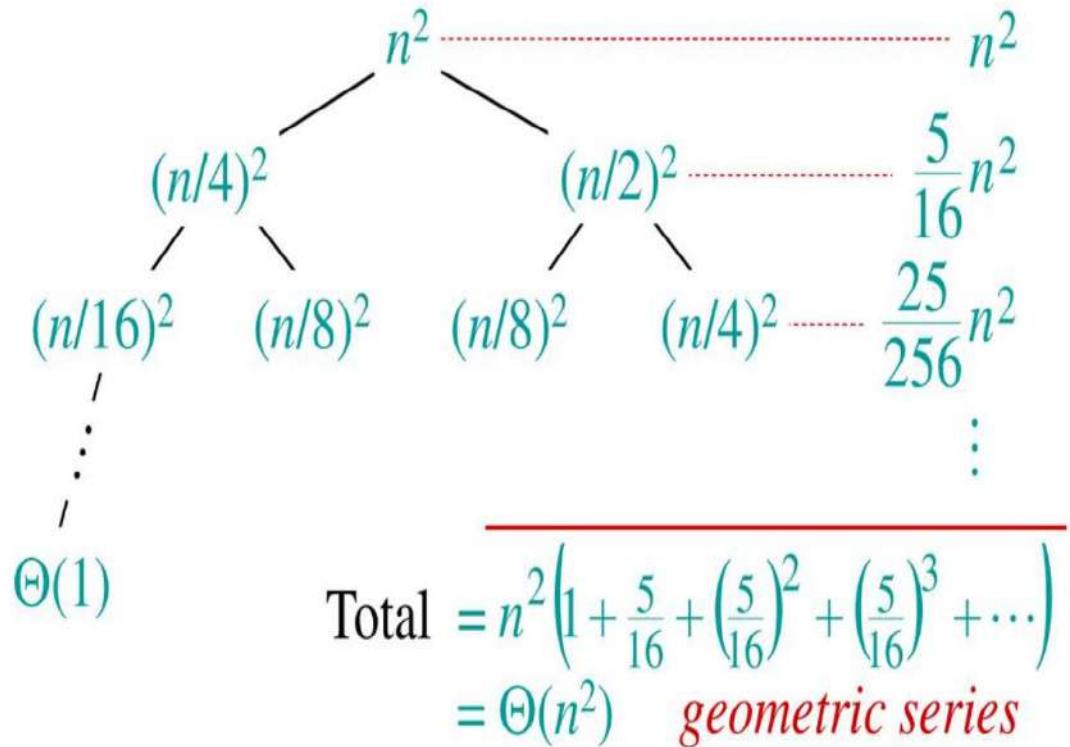
**Solution:**



2. Find the complexity of the below recurrence equation using the tree method

$$T(n) = T(n/4) + T(n/2) + n^2$$

**Solution:**



$$1+x+x^2+\dots+x^n = \frac{1-x^{n+1}}{1-x} \text{ for } x \neq 1$$

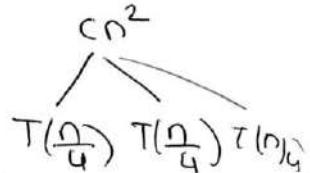
$$1+x+x^2+\dots = \frac{1}{1-x} \text{ for } |x| < 1$$

3. Find the complexity of the below recurrence equation using the tree method

$$T(n) = 3T(n/4) + cn^2$$

**Solution:**

Sol:



$$T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{16}\right) + c\left(\frac{n^2}{4}\right)$$

levels

$$0 \rightarrow \frac{n}{4^0}$$

$$1 \rightarrow \frac{n}{4^1}$$

$$2 \rightarrow \frac{n}{4^2}$$

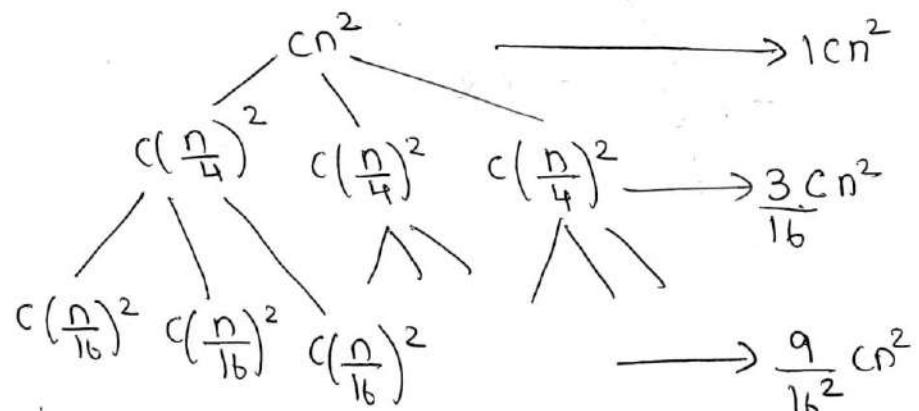
$$\text{level } i = \frac{n}{4^i}$$

$$\frac{n}{4^i} = 1$$

$$n = 4^i$$

Taking log on both sides

$$i = \log_4 n$$



$$\log n = \log 4^i$$

$$\log n = i \log 4$$

$$i = \log_4 n$$

Here ,

$$\text{level } 0 = 3^0 = 1$$

$$\text{level } 1 = 3^1 = 3$$

$$\text{level } 2 = 3^2 = 9$$

$$\vdots$$

$$\text{level } i = 3^i = 3^{\log_4 n}$$

$$= n^{\log_4 3}$$

$$\text{Cost of last level} = n^{\log_4 3} \times T(1)$$

$$= \Theta(n^{\log_4 3})$$

Add cost of all levels

$$= \left[ cn^2 + \frac{3}{16} cn^2 + \frac{9}{16^2} cn^2 + \dots \right] + \Theta(n^{\log_4 3})$$

$$= cn^2 \left[ 1 + \frac{3}{16} + \frac{9}{16^2} + \dots \right] + \Theta(n^{\log_4 3})$$

$$\underbrace{a, ar, ar^2, \dots}_{a, ar, ar^2, \dots}$$

$$S_n = \frac{a}{1-r} - \frac{ar^n}{1-r}$$

$$1 + \frac{3}{16} + \frac{9}{16^2} + \dots$$

$$= \frac{a}{1-r} - \frac{ar^{\infty}}{1-r} \quad r = \frac{a_2}{a_1} = \frac{\frac{3}{16}}{\frac{1}{1}} = \frac{3}{16}$$

$$= \frac{1}{1 - \frac{3}{16}} - \frac{1 \cdot \left(\frac{3}{16}\right)^i}{1 - \frac{3}{16}} \quad i \rightarrow \text{level}$$

$$= \frac{1}{\frac{16-3}{16}} - \frac{\left(\frac{3}{16}\right)^{\log_4 n}}{\frac{16-3}{16}}$$

$$= \frac{16}{13} - \frac{16}{13} \left(\frac{3}{16}\right)^{\log_4 n}$$

$$= \frac{16}{13} \left(1 - \left(\frac{3}{16}\right)^{\log_4 n}\right)$$

$$cn^2 \left[ 1 + \frac{3}{16} + \frac{9}{16^2} + \dots \right] + \Theta(n^{\log_4 3})$$

$$\frac{16}{13} cn^2 \left( 1 - \left(\frac{13}{16}\right)^{\log_4 n} \right) + \Theta(n^{\log_4 3})$$

$$\frac{16}{13} cn^2 - \frac{16}{13} cn^2 \cdot \left(\frac{13}{16}\right)^{\log_4 n} + \Theta(n^{\log_4 3})$$

$$= \Theta(n^2)$$

4. Find the complexity of the below recurrence equation using the back substitution method

$$T(n) = T(n/2) + c \quad \text{if } n > 1$$

$$T(1) = 1 \quad \text{if } n = 1$$

**Solution:**

$$\begin{aligned} T(n) &= T(n/2) + c \longrightarrow ① \\ T\left(\frac{n}{2}\right) &= T\left(\frac{n}{2} \times \frac{1}{2}\right) + c \\ &= T\left(\frac{n}{4}\right) + c \longrightarrow ② \\ T\left(\frac{n}{4}\right) &= T\left(\frac{n}{8}\right) + c \longrightarrow ③ \end{aligned}$$

Backward Substitution.

Substitute ② in ①

$$\begin{aligned} T(n) &= T\left(\frac{n}{4}\right) + c + c \\ &= T\left(\frac{n}{2^2}\right) + 2c \\ &= T\left(\frac{n}{8}\right) + 3c \\ &= T\left(\frac{n}{2^3}\right) + 3c \\ &= T\left(\frac{n}{2^4}\right) + 4c \\ &\xrightarrow{\text{K steps}} \underline{T\left(\frac{n}{2^K}\right) + Kc} \end{aligned}$$

Now, we should use the boundary (base) condition  
(i.e)  $T(1) = 1$

In order to use the boundary condition, the entity inside  $T()$  must be 1.

$$T\left(\frac{n}{2^k}\right) + kC \xrightarrow{\text{To make } T(1)}$$

We assume,

$$n = 2^k \text{ Substitute in } ④$$

$$T\left(\frac{n}{n}\right) + kC$$

$$T(1) + kC$$

$$1 + kC$$

Now, calculate the value of 'k', take log on both sides

$$n = 2^k$$

$$k = \log_2 n$$

$$O(\log n)$$

Logarithmic form	$\log_2 N = k$
Exponential form	$2^k = N$

**5. Find the complexity of the below recurrence equation using the master method**

$$T(n) = T(n) = 2T(n/2) + \sqrt{n}$$

**Solution:**  $T(n) = 2T(n/2) + \sqrt{n}$ , the values of  $a = 2$ ,  $b = 2$  and  $k = 1/2$ .

Here  $\log_b(a) = \log_2(2) = 1 > k$ .

Therefore, the complexity will be  $\Theta(n)$

**VELLORE INSTITUTE OF TECHNOLOGY, VELLORE****QUIZ I****QUESTION PAPER**

---

1. If  $f(n) = 5n + 3$  and  $g(n) = n$  and  $f(n) = O(g(n))$  then the values of  $c$  and  $n_0$  are \_\_\_\_\_.  
a)  $c=6$  and  $n_0=3$    b)  $c=6$  and  $n_0=2$    c)  $c=4$  and  $n_0=1$  d)  $c=5$  and  $n_0=0$
2. What is the time complexity of the following code snippet?

```
fun (int n)
{
    int i=1;
    for (i=1; i*i <=n; i++)
        printf ("Hello\n");
}
```

a)  $O(n)$       b)  $O(n \log n)$       c)  $O(1)$       d)  $O(\sqrt{n})$
3. A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is CORRECT (n refers to the number of items in the queue)?
  - a) Both operations can be performed in  $O(1)$  time.
  - b) At most one operation can be performed in  $O(1)$  time but the worst-case time complexity for the other operation will be  $\Omega(n)$
  - c) The worst-case time complexity for both operations will be  $\Omega(n)$ .
  - d) Worst-case time complexity for both operations will be  $\Omega(\log n)$ .
4. The best data structure to check whether an arithmetic expression has balanced parentheses is a \_\_\_\_\_.  
a) Queue   b) Stack      c) Tree      d) Graph
5. The result of the evaluation of postfix expression is  $8\ 4\ -3\ *1\ 5\ +\ / *$   
a) 14      b) -14      c) -16      d) 18
6. Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following is TRUE?
  - I. Quicksort runs in  $\theta(n^2)$  time.
  - II. Bubble sort runs in  $\theta(n^2)$  time.
  - III. Merge sort runs in  $\theta(n)$  time.
  - IV. Insertion sort runs in  $\theta(n)$  time.  
a) I and IV      b) I , II and III      c) I and II      d) I and III

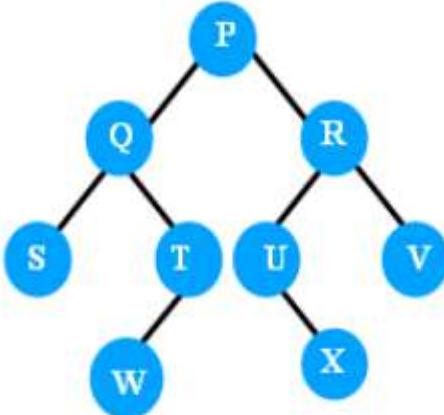


7. The number of swapping needed to sort the numbers 8, 22, 7, 9, 31, 19, 5, and 13 in ascending order using bubble sort is \_\_\_\_\_.  
a) 11      b) 12      c) 13      d) 14
8. Let P be a Quicksort Program to sort numbers in ascending order using the first element as a pivot. Let t1 and t2 be the number of comparisons made by P for the inputs {1,2,3,4,5} and { 4, 1, 5, 3, 2} respectively. which one of the following holds?  
a) t1=6      b) t1 < t2      c) t1>t2      d) t1=t2
9. While Inserting the elements 71, 65, 84, 69, 66, and 83 in an empty binary search tree. The element in the lowest level is \_\_\_\_\_.  
a) 67      b) 65      c) 66      d) 83
10. The height of a tree is defined as the length of the longest root-to-leaf path in it. The maximum & minimum number of nodes in a binary tree of height 5 is \_\_\_\_\_.  
a) 63, 6      b) 64, 5      c) 32, 6      d) 31, 5

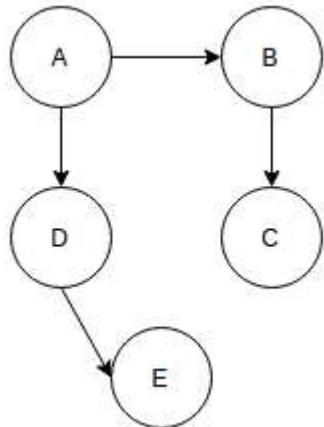
**VELLORE INSTITUTE OF TECHNOLOGY, VELLORE****QUIZ II****QUESTION PAPER**

---

1. Given an array of element 5, 7, 9, 1, 3, 10, 8, 4. Which of the following is the correct sequences of elements after inserting all the elements in a min-heap?  
(a) 1,4,3,9,8,5,7,10  
(b) 1,3,4,5,8,7,9,10  
(c) 1,3,7,4,8,5,9,10  
**(d) 1,3,4,5,7,8,9,10**
2. Find the postorder traversal of the binary tree shown below.



- (a) P Q R S T U V W X  
(b) W R S Q P V T U X  
**(c) S W T Q X U V R P**  
(d) S T W U X V Q R P
3. What would be the DFS traversal of the given Graph?





- (a) ABCED  
(b) AEDCB  
(c) EDCBA  
**(d) ADEBC**
4. The keys 36,18, 72, 43, 6, 10, 5 and 15 are inserted into an initially empty hash table of length 8 using linear probing with hash function  $h(k) = k \bmod \text{tablesize}$ . Find the location of each value in the resultant hash table?  
(a) 4,2,0,3,6,2,5,7  
**(b) 4,2,0,3,6,5,7,1**  
(c) 4,2,0,3,6,5,1,7  
(d) 4,2,0,3,6,1,5,7  
(e) 4,2,0,3,6,1,7,5  
(f) 4,2,0,3,6,7,5,1  
(g) 4,2,0,3,6,7,1,5
5. A graph in which all vertices have equal degree is known as...?  
(a) Regular graph  
(b) Multi graph  
(c) Simple graph  
**(d) Complete graph**
6. To implement Dijkstra's shortest path algorithm on unweighted graphs so that it runs in linear time, the data structure to be used is:  
**(a) Queue**  
(b) Stack  
(c) Heap  
(d) B Tree
7. The keys 18,41,22,44,59,32,31 and 73 are inserted into an initially empty hash table of length 13 using Double Hashing with hash function  $h(k) = k \bmod \text{tablesize}$  and  $d(k)=7-k \bmod 7$ . Find the location of each value in the resultant hash table?  
**(a) 5, 2, 9, 10, 7, 6, 0, 8**  
(b) 5, 2, 9, 10, 7, 6, 8, 0  
(c) 5, 2, 9, 10, 6, 7, 8, 0  
(d) 5, 2, 9, 10, 6, 7, 0, 8  
(e) 5, 2, 9, 4, 7, 6, 5, 8



8. In delete operation of AVL, we need inorder successor (or predecessor) of a node when the node to be deleted has both left and right child as non-empty. Which of the following is true about inorder successor needed in delete operation?
  - (a) Inorder Successor is always a leaf node
  - (b) Inorder successor is always either a leaf node or a node with empty left child
  - (c) Inorder successor may be an ancestor of the node
  - (d) Inorder successor is always either a leaf node or a node with empty right child
9. The pre-order and in-order are traversals of a binary tree are T M L N P O Q and L M N T O P Q. Which of following is post-order traversal of the tree?
  - (a) L N M O Q P T
  - (b) N M O P O L T
  - (c) L M N O P Q T
  - (d) O P L M N Q T
10. The time complexity of computing the transitive closure of a binary relation on a set of n elements is known to be:
  - (a) O(n)
  - (b) O(nLogn)
  - (c) O( $n^{(3/2)}$ )
  - (d) O( $n^3$ )

D1  
Seat No.



VIT<sup>®</sup>

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)



## Test : CAT- I / CAT- II / Mid-Term

Register No.: 22BCB0001 Course Code: BCS.E.2024 Class NBR: 6.3.3.1

Student's Name: Mr/Ms. K. Geethika Varshini Course Title: Data Structures & Algorithms

Programme : B.TECH School : SCOPE Faculty Name : Prof. Priyanka N

Slot : A Session : FN/AN/EVN Date of Exam : 26/3/23 No. of pages written :

Signature of the Invigilator :

P. Ray -

Name / Emp. ID of the Invigilator : 17338

K. Geethika  
Signature of the Student

Q No.	Marks										
	a	b		a	b		a	b		a	b
1	4	2	9			17			25		33
2	3	5	10			18			26		34
3	8		11			19			27		35
4	8		12			20			28		36
5	X		13			21			29		37
6			14			22			30		38
7			15			23			31		39
8			16			24			32		40
Sub-Total	30	X	Sub-Total			Sub-Total			Sub-Total		Sub-Total

Grand Total Marks :

37

Evaluator's Signature

Date : .....

N. Ray

Evaluator's Name : Prof. PRITHYANKA N

Marks In Words ..... THREE SEVEN .....

i) a) i)  $T(n) = 3T\left(\frac{n}{2}\right) + n^2$

Master's theorem:  $aT(n/b) + f(n)$

$$f(n) = n^k \log^p n$$

$$a = 3$$

$$b = 2$$

$$\log_b^a = \log_2^3 = 1.5$$

$$f(n) = n^2 \log n$$

$$K = 2, P = 0$$

$$\log < K, \text{ so}$$

~~$$\text{if } P \geq 0 = O(n^k \log^p n)$$~~

~~$$P \leq 0 = O(n^k)$$~~

~~$$\text{As, } P = 0$$~~

~~$$O(n^2 \log n)$$~~

ii)  $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$

Master's theorem:  $aT(n/b) + f(n)$

$$f(n) = n^k \log^p n$$

$$a = 2$$

$$b = 2$$

$$\log_b^a = \log_2^2 = 1$$

$$f(n) = n' \log'n$$

$$K=1, P=1$$

Here,  $\log = K$

So,

if  $\log = K$

$$P > -1 = O(n^k \log^{1+p} n)$$

$$P = -1 = O(n^k \log \log n)$$

$$P < -1 = O(n^k)$$

As,  $P = 1$

~~$$O(n^k \log^2 n)$$~~

b) ii)  $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1, & \text{for } n > 1 \\ , & \text{for } n = 1 \end{cases}$

$$T(n) = T\left(\frac{n}{2}\right) + 1 \quad \dots \textcircled{1}$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1 \quad \dots \textcircled{2}$$

sub eq \textcircled{2} in eq \textcircled{1}

$$T(n) = T\left(\frac{n}{4}\right) + 1 + 1$$

$$T(n) = T\left(\frac{n}{8}\right) + 2$$

$$T(n) = T\left(\frac{n}{16}\right) + 3$$

$$T(n) = T\left(\frac{n}{32}\right) + 4$$

$$\vdots$$

$$\vdots$$

$$\vdots$$

$$\vdots$$

$$\vdots$$

$$\vdots$$

$$\vdots$$

$$\begin{aligned} T(n) &= \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \dots \\ &= n \left( \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots \right) \end{aligned}$$

$$2^K = n$$

$$K \log_2 2$$

$$2^k = n$$

log on both sides

$$K \log_2 = \log n$$

$$K = \log n$$

$$O(n) = O(n \log n)$$

3.  $(a/(b-c+d))^*(e-a)^*c$

Input	Stack	Postfix / Output
(	(	
a	(a	a
/	(/a	a
(	(/a(	a
b	(/a(b	ab
-	(/a(b-	ab
c	(/a(b-c	abc
+	(/a(b-c+	abc
d	(/a(b-c+d	abcd
)	(/a(b-c+d)	abcd+-
)		abcd+/-

*	*	abcd+/-
(	*(	abcd+/-/
e	*(	abcd+/-/e
-	*(-	abcd+/-/e
a	*(-	abcd+/-/ea
)	*	abcd+/-/ea-
*	**	abcd+/-/ea-
c	**	abcd+/-/ea-c

Result  $\rightarrow$  abcd+/-/ea-c \*\*

### Algorithm

- When an operand comes we push the operand into the postfix
- When an operator comes we push the operator into the stack
- When an operator has higher precedence than top operator in the stack, we push the operator into stack
- When an operator has same precedence than top operator in stack, we push the operator into the stack

→ When the operator has lower precedence than the top operator in the stack, we pop the operator present in stack until we get operator of same precedence or lower precedence.

e.g. Here in this question we can see stack has  $(/(- +$  operator is added & the stack becomes  $(/- +$  becoz they have same precedence.

→ When ever we see an open parenthesis ' $($ ' we push it into the stack

$\wedge$  - higher precedence  
 $*$ / $\%$  - medium precedence  
 $+, -,$  lower precedence

→ When ever we see an closed parenthesis we pop all the operators present in the stack to postfix.

e.g. Here in the above question we can see stack has  $(/- + )'$  closed parenthesis is added, so we pop the operators present in the open parenthesis & becomes  $/$ .

→ At the end when operators are left in the stack we pop all the operators & we can place them in final ending of equation.

4. a) Advantages of circular queue

→ As circular queue helps in efficient memory storage.

→ It takes less time than linear queue.

→ In linear queue once an element is deleted the memory utilised by the deleted element cannot be used again by new element i.e., we cannot add new element to the location of deleted element. but in circular queue we can add new element & we can even push more the elements.

→ The linear queue is in linear form & the circular queue is in the form of circle as the circular queue is in the form of circle it takes less time than linear queue.

6) f

For  
REAR = MAX - 1  
set OVERFLOW

If REAR = -1, FRONT = -1

SET REAR = FRONT = 0

y

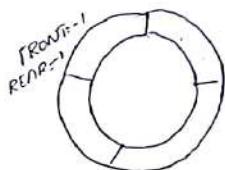
else if  
REAR != 0, FRONT != 0 - 1  
SET REAR = 1 + REAR

else

SET VAL = enqueue[REAR]

initial we have to check whether the queue is empty. If it is full is overflow.

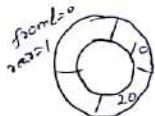
→ As,  $N=4$  , circular queue



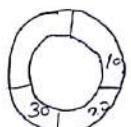
→ Initially insert the element 10,  $front=0$ ,  $rear=0$



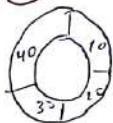
→ then insert the element 20,  $front=0$ ,  $rear=1$



→ then insert the element 30,  $front=0$ ,  $rear=2$

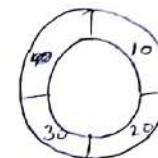


→ then insert the element 40,  $front=0$ ,  $rear=3$

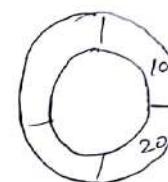


2. Deletion of two elements

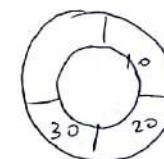
→ we delete the elements from front end.  
 $rear=3$ ,  
 $front=1$



→ then we delete other element,  $rear=3$ ,  $front=2$



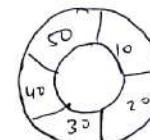
3. p insert the element 30,  $rear=4$ ,  $front=2$



insert the element 40,  $rear=5$ ,  $front=2$

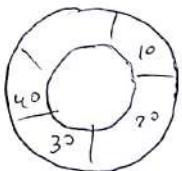


Here are going to insert other extra element 50  
 $rear=6$ ,  $front=2$

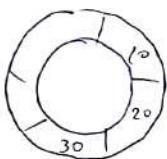


4. Successive deletion of three elements

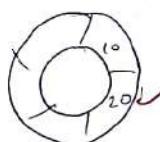
→ First we delete the top element in front end. Rear=6, front=3



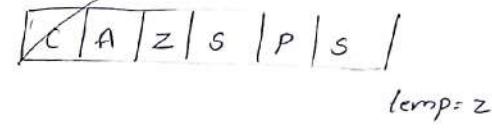
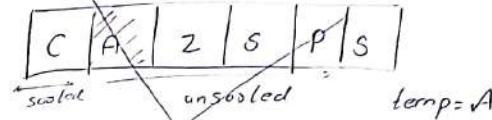
→ then we delete the preceding element in front end. Rear=6, front=4



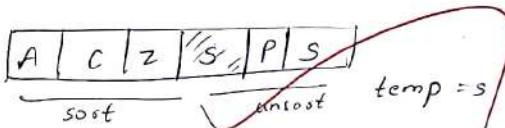
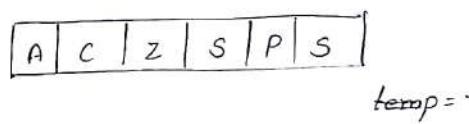
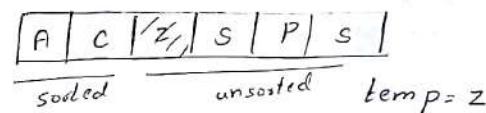
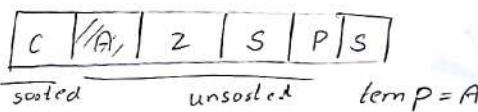
→ then we delete the next preceding element  
Rear=6, front=5



5. Insertion sort



6. Insertion sort



A	c	s	z	P	s
sort		unsort		temp = P	

A	c	s	z	z	s
---	---	---	---	---	---

A	c	p	s	z	s
---	---	---	---	---	---

A	c	p	s	z	S
sort		unsort		temp = S	

A	c	p	s	S	z
---	---	---	---	---	---

```

for (i=0, i<n-1, i++)
int main()
{
    for (j=1-i, j<i, j++)
        while (Array[a] < min[i])
            if (Array[a] < min[j])
                min[j]
            if (Array[a] > min[i])
                {
                    swap (min, a) values
                }
            }
    return 0
}

```

2. b)  $f(n) = 3n + 2$

in big-O  $O > f(n) > g(n)$

$$O > 3n+2 \rightarrow 3n+2n$$

$$O > 3n+2 > 3n+2n, \text{ if } n=1$$

$$O > 5 >$$

$$O > 3n+2 > 3n+4n$$

$$\text{if } n=1$$

$$O > 3n+5 > 7$$

if satisfy the above condition

$$f(n) = O(g(n))$$

$$O > f(n) > c \cdot g(n)$$

$$c = 2$$

$$\& n_0 = 1$$

∴ The values of  $c=2$

and the value of  $n_0=1$

2. a) i)  $\text{sum} = 0 \quad \text{---} \quad 1$

$\text{for } (k=1; k \leq n; k=k+2) \rightarrow O(n+1)$

$\text{for } (j=1; j < n; j++) \rightarrow O(n)$

$\text{sum} += n \times n$

$k \quad 1 \quad 2 \quad 4 \quad 8 \quad 16 \quad \rightarrow 2^k$

$j \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \rightarrow k$

~~n~~  $n = 2^k$

$$\log n = k \log_2 2$$

$$k = \log n$$

$$\therefore O(\log n)$$

b) ii)  $\text{sum} = 0;$

$\text{for } (j=1; j < n; j++)$

$\text{for } (i=1; i < j; i++)$

$\text{sum}++;$

$\text{for } (k=0; k < n, k++)$

$$a[k] = k$$

$i \quad 1 \quad 2 \quad 3 \quad \dots \quad n$

$j \quad 1 \quad 2 \quad 3 \quad \dots \quad n$

$k \quad 0 \quad 1 \quad 2 \quad \dots \quad n-1$

$$\therefore O(\Theta(n^3))$$