

# Data Structure Using C

Mr. Rajesh Pandey

**Asst. Professor**

**Shobhit Institute of Engineering & Technology  
(Deemed-to-be-University), Meerut**

Email: [rajesh@shobhituniversity.ac.in](mailto:rajesh@shobhituniversity.ac.in)

Mobile: 7417970699

# CONTENTS

1. Introduction to Data Structure and Algorithm
2. Array.
3. Stack.
4. Infix, Prefix and Postfix expression
5. Queue
6. Pointers.
7. Linked list
8. Tree traversal
9. Binary search tree

# 1. Introduction to Data Structure and Algorithm

- Data Structure: Definition.
- Data Structure: Types.
- Need of Data Structure.
- Application of Data Structure.
- Algorithm: Definition
- Characteristics of Algorithm

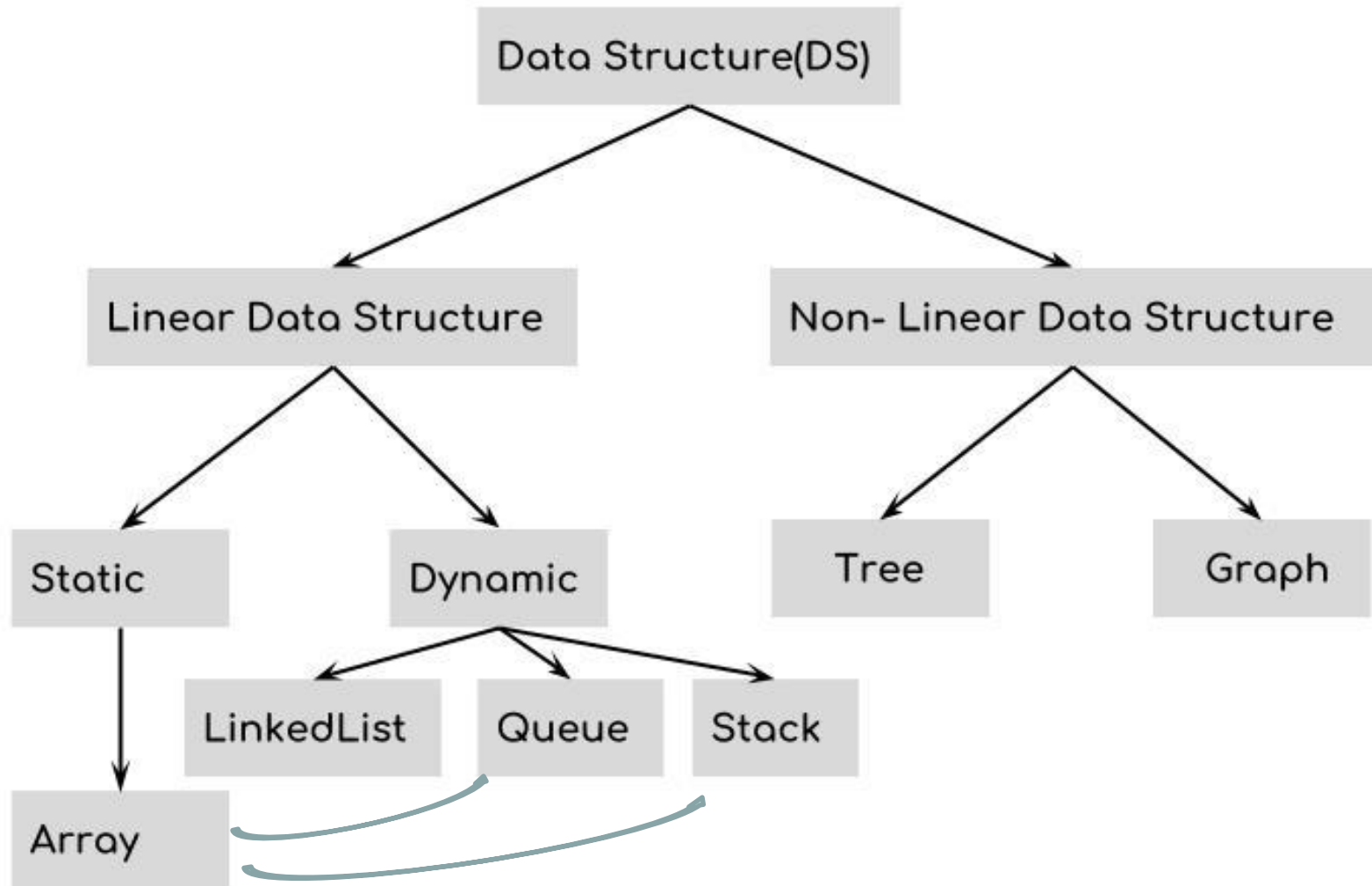
# Data Structure: Definition

- A **data structure** is a special way of organizing and storing data in a computer so that it can be used efficiently.
- Array, Linked List, Stack, Queue, Tree, Graph etc are all data structures that stores the data in a special way so that we can **access and use the data efficiently**.
- Each of these mentioned **data structures** has a different special way of organizing data so we choose the data structure based on the requirement.

# Data Structure: Types

- We have two types of data structures:
  1. Linear Data Structure
  2. Non-linear Data Structure
- **Linear data structures:** Elements of Linear data structure are accessed in a sequential manner, however the elements can be stored in these data structure in any order. Examples of linear data structure are: LinkedList, Stack, Queue and Array.
- **Non-linear data structures:** Elements of non-linear data structures are stores and accessed in non-linear order. Examples of non-linear data structure are: Tree and Graph

# Data Structure: Types



# Why Data Structures are needed?

- With increasing complexities in computer algorithms, the amount of data usage is increasing, this can affect the performance of the application and can create some areas of concern:
  - To handle very large data,
  - To achieve high-speed processing.
  - Data Search: Getting a particular record from database should be quick and with optimum use of resources.
  - Multiple requests: To handle simultaneous requests from multiple users

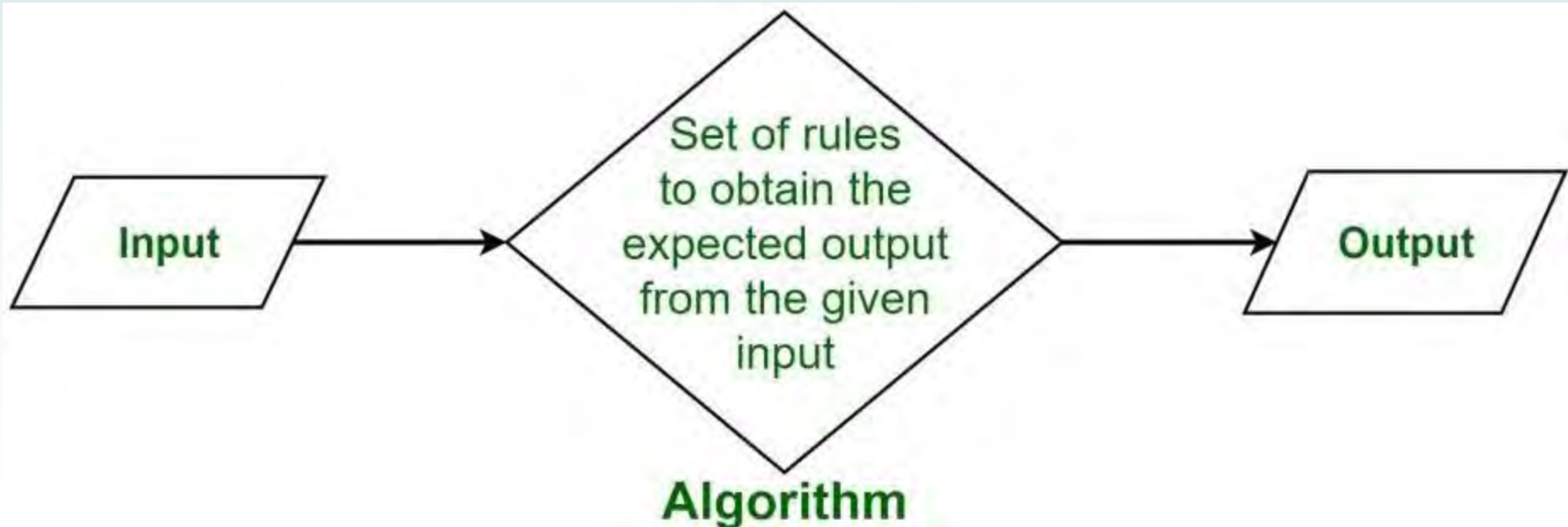
# Advantages of Data Structures

- **Efficient Memory use:** With efficient use of data structure memory usage can be optimized, for e.g we can use linked list vs arrays when we are not sure about the size of data. When there is no more use of memory, it can be released.
- **Reusability:** Data structures can be reused, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.



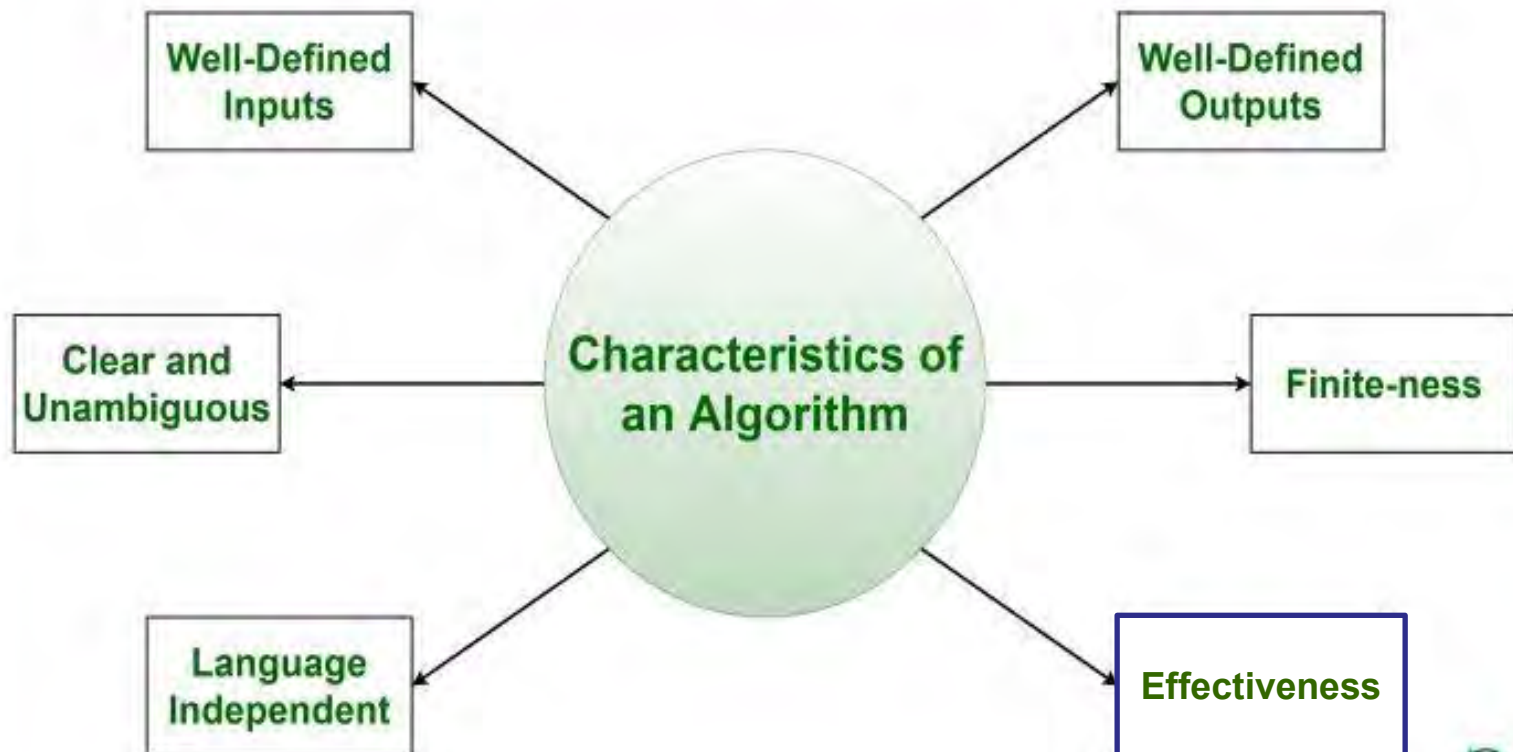
# Algorithm

- Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output.
- An algorithm are generally analyzed on two factors – time and space. That is, how much execution time and how much extra space required by the algorithm.



# Characteristics of Algorithm

## Characteristics of an Algorithm



# Characteristics of Algorithm

An algorithm must have the following characteristics:

- 1.Clear and Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- 2.Well-Defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs.
- 3.Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- 4.Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

# Characteristics of Algorithm

5. **Effectiveness:** It is measured in terms of time and space. An algorithm must require minimum memory space and should have minimum execution time for its successful execution.
6. **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

## 2. Arrays

- Array: Definition.
- Advantages & Disadvantages of Array
- Application of Arrays.
- Basic Operations on Array.
  - ✓ Insertion & Traversal of Array
  - ✓ Deletion from Array
  - ✓ Searching in Array
    - Sequential Search
    - Binary Search
- Two Dimensional Array

# Arrays in 'C'

- An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations. Here the words,
  - **finite** *means* data range must be defined.
  - **ordered** *means* data must be stored in continuous memory addresses.
  - **homogenous** *means* data must be of similar data type.
- Following are the important terms to understand the concept of Array.
  - **Element** – Each item stored in an array is called an element.
  - **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

# Advantages & Disadvantages of Array

- **Advantages :**

- It is used to represent multiple data items of same type by using only single name.
- It can be used to implement other data structures like stacks, queues, trees, graphs etc.
- 2D arrays are used to represent matrices.
- It is easy to understand and implement.

# Advantages & Disadvantages of Array

- **Disadvantages :**

- We must know in advance that how many elements are to be stored in array.
- Array is static structure. It means that array is of fixed size. The memory which is allocated to array can not be increased or reduced.
- Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. And if we allocate less memory than requirement, then it will create problem.
- Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations and the shifting operation is costly.



# Applications of Array

- Arrays can be used for sorting data elements. Different sorting techniques like Bubble sort, Insertion sort, Selection sort etc use arrays to store and sort elements easily.
- Arrays can be used for performing matrix operations. Many databases, small and large, consist of one-dimensional and two-dimensional arrays whose elements are records.
- Arrays can be used for CPU scheduling.
- Lastly, arrays are also used to implement other data structures like Stacks, Queues, Heaps, Hash tables etc

# Basic Operations

- Following are the basic operations supported by an array.
  - **Traverse** – print all the array elements one by one.
  - **Insertion** – Adds an element at the given index.
  - **Deletion** – Deletes an element at the given index.
  - **Search** – Searches an element using the given index or by the value.
  - **Update** – Updates an element at the given index.

# Insertion & Traversal Operation

```
void main()
{
    int arr[4];
    int i, j;
    printf("Enter array element");
    for(i = 0; i < 4; i++)
    {
        scanf("%d", &arr[i]);    //Run time array initialization
    }
    for(j = 0; j < 4; j++)
    {
        printf("%d\n", arr[j]);
    }
}
```

# Deletion Operation

```
main() {  
    int arr[] = {1,3,5,7,8};  
    int k = 3, n = 5; // 3rd element has to be deleted.  
    int i , j ;  
    printf("The original array elements are :\n");  
    for(i = 0; i<n; i++)  
    {  
        printf("arr[%d] = %d \n", i, arr[i]);  
    }  
    j = k ;  
    while( j < n)  
    {  
        arr[ j -1 ] = arr[ j ];  
        j = j + 1;  
    }  
    n = n - 1 ;  
    printf("The array elements after deletion :\n");  
    for(i = 0; i<n; i++) {  
        printf("arr[%d] = %d \n", i, arr[i]);  
    }  
}
```

## OUTPUT

The original array elements are:

arr[0] = 1

Arr[1] = 3

Arr[2] = 5

Arr[3] = 7

Arr[4] = 8

The array elements after deletion:

arr[0] = 1

Arr[1] = 3

Arr[2] = 7

Arr[3] = 8

# Searching in Array

## What is Searching?

- Searching is the process of finding the position of a given value in a list of values.
- It decides whether a search key is present in the data or not.

## Searching Techniques:

- To search an element in a given array, it can be done in following ways:
  1. Sequential/Linear Search
  2. Binary Search

# Sequential Search

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it outputs “value not present”.



Fig. Sequential Search

# Binary Search

- Binary Search is used for searching an element in a sorted array.
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.



5	10	15	20	25	30
---	----	----	----	----	----

- The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

# Binary Search

- For example:



Fig. Working Structure of Binary Search

- Binary searching starts with middle element. If the middle element is equal to the element that we are searching then return true. If the middle element is less than then move to the right of the list or if the middle element is greater then move to the left of the list. Repeat this, till you find an element.



# Two Dimensional Array

```
#include<stdio.h>

int main(){
    /* 2D array declaration*/
    int abc[5][4];
    /*Counter variables for the loop*/
    int i, j;
    for(i=0; i<5; i++) {
        for(j=0;j<4;j++) {
            printf("Enter value for abc[%d][%d]:", i, j);
            scanf("%d", &abc[i][j]);
        }
    }
    return 0;
}
```

- i = 0  
J=0 to 3

## 2D array conceptual memory representation

Second subscript →

first subscript ↓

abc[0][0]	abc[0][1]	abc[0][2]	abc[0][3]
abc[1][0]	abc[1][1]	abc[1][2]	abc[1][3]
abc[2][0]	abc[2][1]	abc[2][2]	abc[2][3]
abc[3][0]	abc[3][1]	abc[3][2]	abc[3][3]
abc[4][0]	abc[4][1]	abc[4][2]	abc[4][3]

Here my array is abc[5][4], which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means abc[0][0] would be the first element of the array.

# Two Dimensional Array

```
#include<stdio.h>
int main(){
    /* 2D array declaration*/
    int disp[2][3];
    /*Counter variables for the loop*/
    int i, j;
    for(i=0; i<2; i++) {
        for(j=0;j<3;j++) {
            printf("Enter value for disp[%d][%d]:", i, j);
            scanf("%d", &disp[i][j]);
        }
    }
    //Displaying array elements
    printf("Two Dimensional array elements:\n");
    for(i=0; i<2; i++) {
        for(j=0;j<3;j++) {
            printf("%d", disp[i][j]);
            if(j==2){
                printf("\n");
            }
        }
    }
    return 0;
}
```

# Two Dimensional Array

```
#include<stdio.h>
int main(){
    /* 2D array declaration*/
    int disp[2][3];
    /*Counter variables for the loop*/
    int i, j;
    for(i=0; i<2; i++) {
        for(j=0;j<3;j++) {
            printf("Enter value for disp[%d][%d]:", i, j);
            scanf("%d", &disp[i][j]);
        }
    }
    //Displaying array elements
    printf("Two Dimensional array elements:\n");
    for(i=0; i<2; i++) {
        for(j=0;j<3;j++) {
            printf("%d ", disp[i][j]);
            if(j==2){
                printf("\n");
            }
        }
    }
    return 0;
}
```



Enter value for disp[0][0]:1  
Enter value for disp[0][1]:2  
Enter value for disp[0][2]:3  
Enter value for disp[1][0]:4  
Enter value for disp[1][1]:5  
Enter value for disp[1][2]:6  
Two Dimensional array elements:  
1 2 3  
4 5 6

i= 1, J=0

# 3. Stack

- Stack: Definition.
- Application of Stack
- Implementation of Stack
- Basic Operation on Stack
  - PUSH Operation (Insertion)
  - POP Operation (Deletion)

# Stack

## What is Stack?

- Stack is an ordered list of the same type of elements.
- It is a linear list where all insertions and deletions are permitted only at one end of the list.
- Stack is a LIFO (Last In First Out) structure.
- In a stack, when an element is added, it goes to the top of the stack.

## Definition:

“Stack is a collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle”.

# Applications of stack

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers.
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem.

# Implementation of Stack

- Empty stack: **Top = -1** & for full stack: **Top = Max\_Size - 1**

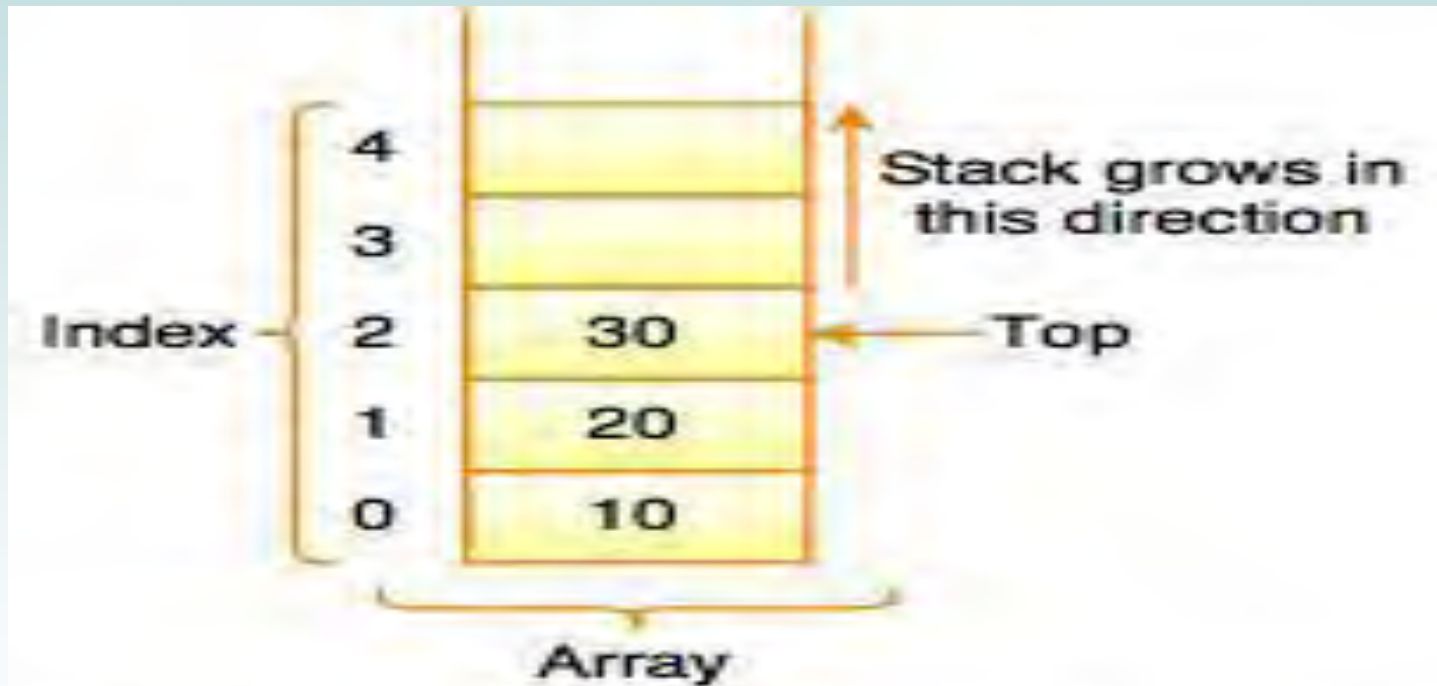


Fig. Implementation Stack using Array

# Stack Overflow & Underflow Condition

- **Stack Overflow** : When the stack is full and user still try to push an element in, the condition is called stack overflow/ stack full.

$$\text{TOP} = \text{Max\_Size} - 1$$

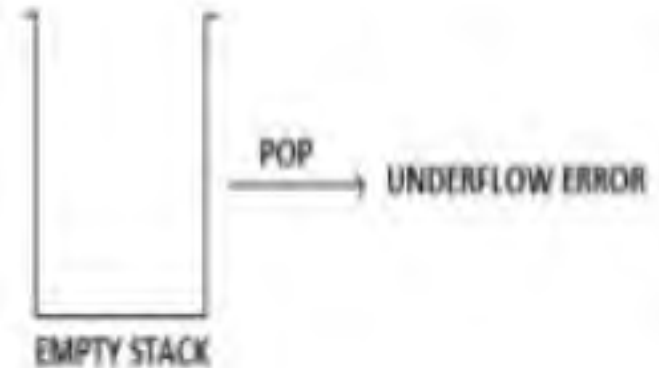
- **Stack Underflow** : When the stack is empty and an user try to pop element from the stack, the condition is called stack underflow/ stack empty.

$$\text{TOP} = -1$$

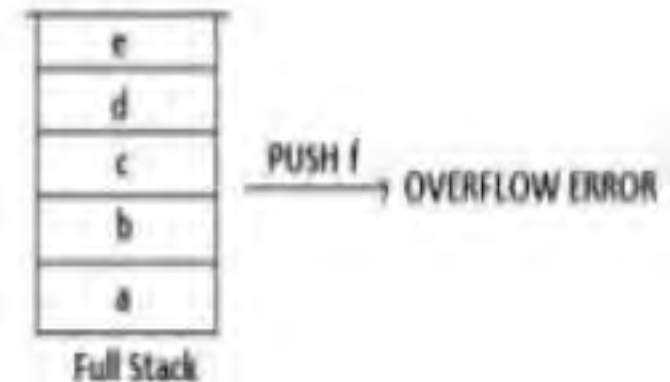


# Stack Overflow & Underflow Condition

- **Stack underflow** happens when we try to pop (remove) an item from the stack, when nothing is actually there to remove.

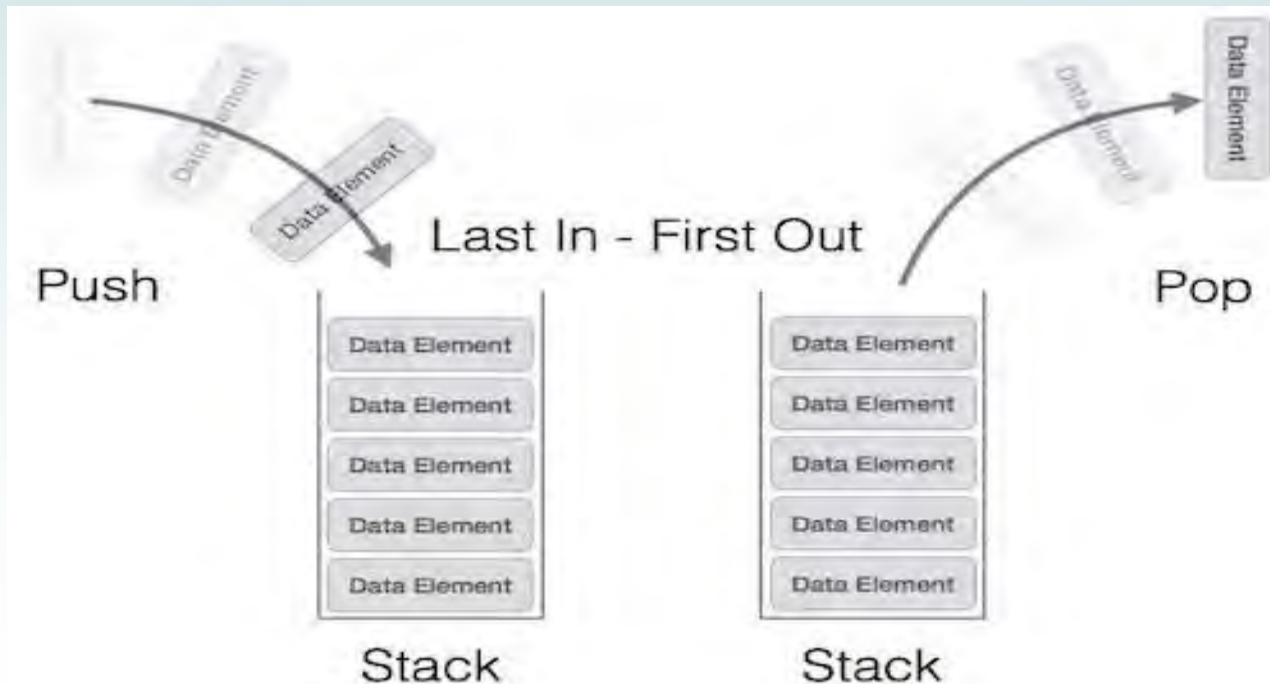


- **Stack overflow** happens when we try to push one more item onto our stack than it can actually hold.



# Basic Operations

- a stack is used for the following two primary operations –
  - **push()** – Pushing (storing) an element on the stack.
  - **pop()** – Removing (accessing) an element from the stack.



# Push Operation

- The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

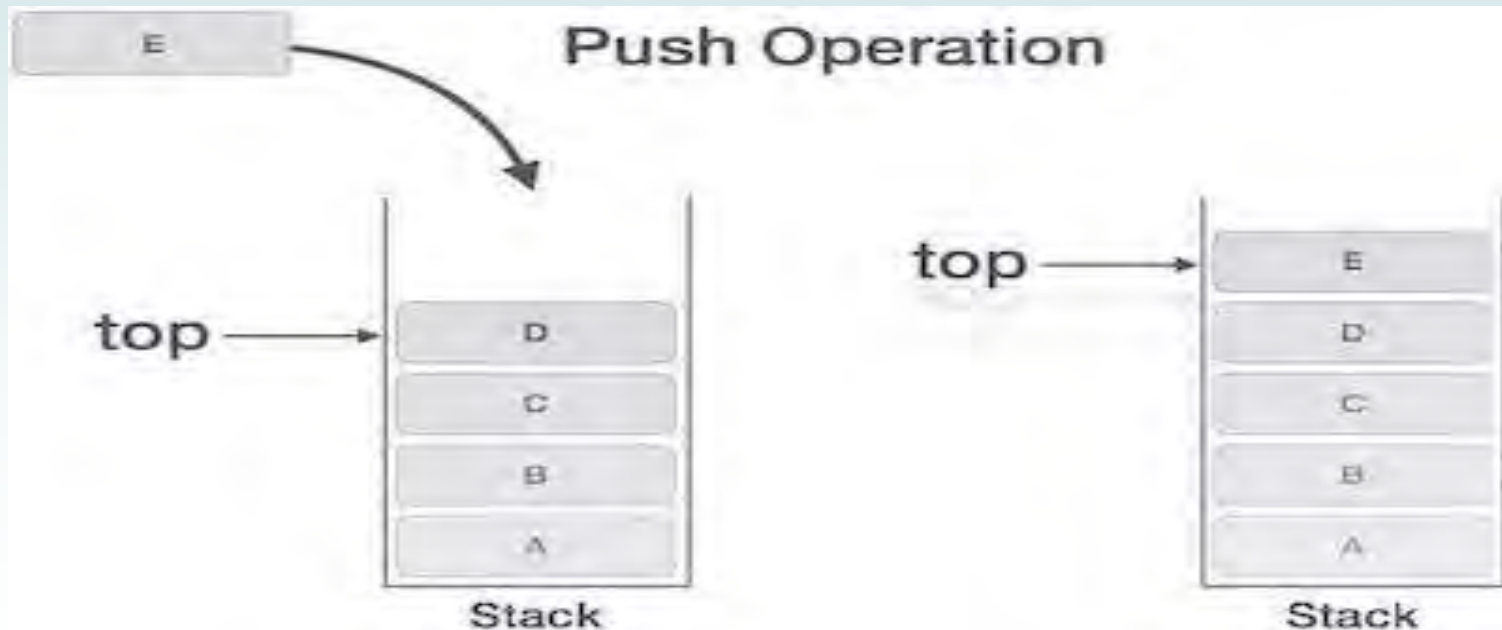
**Step 1** – Checks if the stack is full.

**Step 2** – If the stack is full, output “Stack Overflow”.

**Step 3** – If the stack is not full, increments **top** to point next empty space.

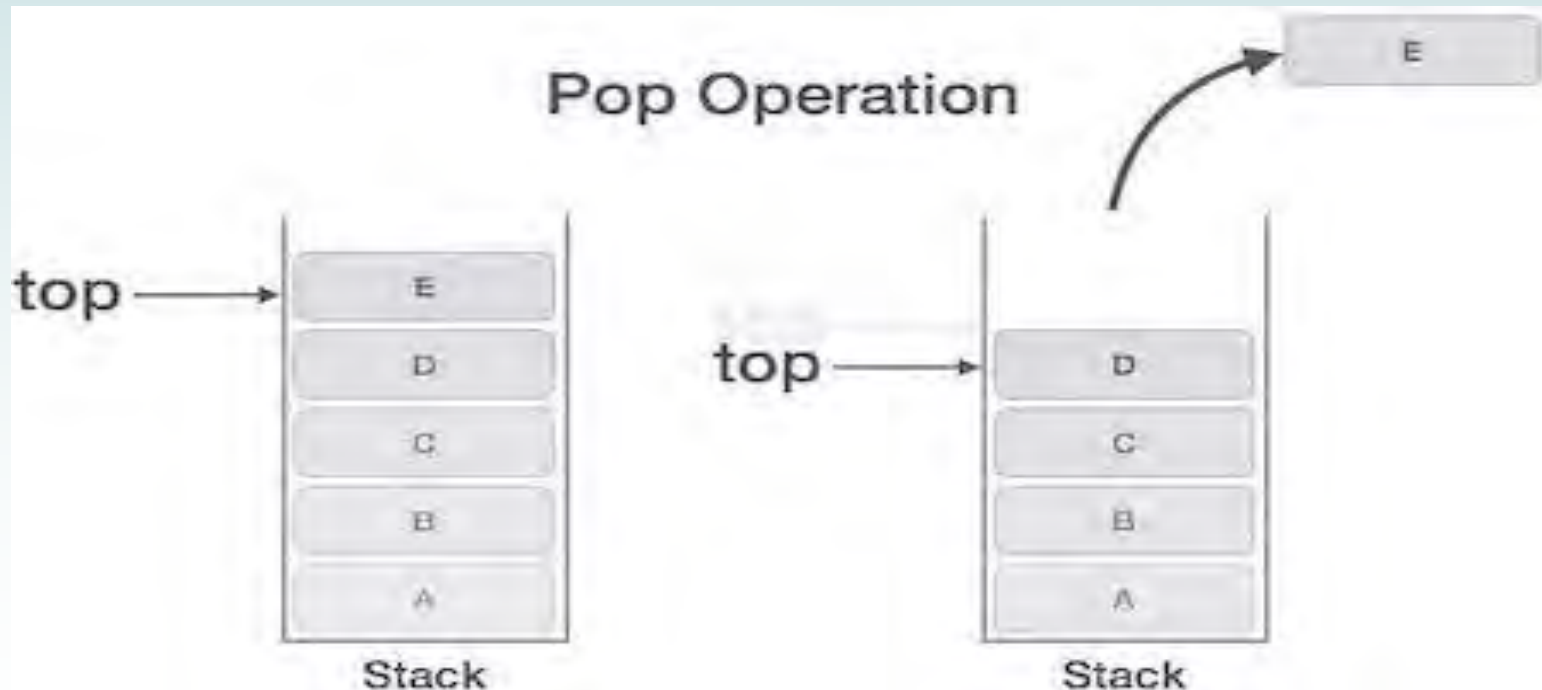
**Step 4** – Adds data element to the stack location, where top is pointing.

**Step 5** – Returns success.



# POP Operation

- A Pop operation may involve the following steps –
  - Step 1** – Checks if the stack is empty.
  - Step 2** – If the stack is empty, produces an error and exit.
  - Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
  - Step 4** – Decreases the value of top by 1.
  - Step 5** – Returns success.



# Program to Implement Stack

```
#include <stdio.h>
#include <conio.h>
void push();
void pop();
void display();
int stack[10], top=-1, element;
void main()
{
    int ch;
    do
    {
        printf("\n\n\n 1. Insert\n 2. Delete\n 3. Display\n 4. Exit\n");
        printf("\n Enter Your Choice: ");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1: push();
                     break;
            case 2: pop();
                     break;
            case 3: display();
                     break;
            case 4: exit ();
            default: printf("\n\n Invalid entry. Please try again...\n");
        }
    }
    while(ch!=4);
    getch();
}
```

# Program to Implement Stack

```
void push()
{
    if(top == 9)
        printf("\n\n Stack is Full.");
    else
    {
        printf("\n\n Enter Element: ");
        scanf("%d", &element);
        top++;
        stack[top] = element;
        printf("\n\n Element Inserted = %d", element);
    }
}
```

```
1. Insert
2. Delete
3. Display
4. Exit
```

```
Enter Your Choice: 1
```

```
Enter Element: 30
```

```
Element Inserted = 30
```

```
1. Insert
2. Delete
3. Display
4. Exit
```

```
Enter Your Choice:
```

# Program to Implement Stack

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

```
1. Insert
2. Delete
3. Display
4. Exit

Enter Your Choice: 3

30
20
10

1. Insert
2. Delete
3. Display
4. Exit

Enter Your Choice: _
```

# Program to Implement Stack

```
void pop()
{
    if(top == -1)
        printf("\n\n Stack is Empty.");
    else
    {
        element = stack[top];
        top--;
        printf("\n\n Element Deleted = %d", element);
    }
}
```

```
1. Insert
2. Delete
3. Display
4. Exit
```

Enter Your Choice: 2

Element Deleted = 30

```
1. Insert
2. Delete
3. Display
4. Exit
```

Enter Your Choice: 3

20

10



# 4. Infix, Prefix and Postfix expression

- Polish notations.
- Infix, Postfix, Prefix form.
- Conversion without stack.
- Conversion with stack.

# Infix, Prefix & Postfix Notations

- An arithmetic expression can be represented in various forms such as Infix, Postfix and Prefix.

Infix Expression	Prefix Expression	Postfix Expression
$A + B$	$+ A B$	$A B +$

- Infix notation is also called as Polish notation and postfix is known as Reverse polish notation.

# Conversion from infix to postfix form

**Example:** Convert the expression  $(A + B) / (C - D)$  to postfix form.

**Solution :**  $(A + B) / (C - D)$

➡  $(AB+) / (CD-)$

Now, Put  $X = (AB+)$  and  $Y = (CD-)$

➡  $X / Y$

➡  $XY/$

Put the value of  $X$  and  $Y$ , we get

➡ **AB+CD-/**

# Conversion from infix to postfix form

**Exercise:** Convert the expression  $A * B + C / D$  to postfix form.

# Conversion from infix to postfix form

**Example:** Convert the expression  $A * B + C / D$  to postfix form.

**Solution :**  $A * B + C / D$

➡  $A * B + CD/$

➡  $A * B + Y$  [ Put  $Y = CD/$ ]

➡  $AB^* + Y$

➡  $X + Y$  [ Put  $X = AB^*$ ]

➡  $XY+$

Put the value of  $X$  and  $Y$ , we get

➡  **$AB^*CD/+$**

# Infix to postfix Conversion Using Stack

## Algorithm:

1. Scan the input string (infix notation) from left to right.
2. If the symbol scanned is an operand, it may be immediately appended to the postfix string.
3. If the next symbol is an operator,
  - Pop and append to the postfix string every operator on the stack that
    - is above the most recently scanned left parenthesis, and
    - has precedence higher than or is a right-associative operator of equal precedence to that of the new operator symbol.
  - Push the new operator onto the stack.
4. When a left parenthesis is seen, it must be pushed onto the stack.
5. When a right parenthesis is seen, all operators down to the most recently scanned left parenthesis must be popped and appended to the postfix string. Furthermore, this pair of parentheses must be discarded.
6. When the infix string is completely scanned, the stack may still contain some operators. All the remaining operators should be popped and appended to the postfix string.

# Infix to postfix Conversion Using Stack

**Example 1:** Suppose we are converting  $3*3/(4-1)+6*2$  expression into postfix form.

Following table shows the evaluation of Infix to Postfix:

Expression	Stack	Output
3	Empty	3
*	*	3
3	*	33
/	/	33*
(	/(	33*
4	/(	33*4
-	/(-	33*4
1	/(-	33*41
)	-	33*41-
+	+	33*41-/
6	+	33*41-/6
*	+	33*41-/62
2	+	33*41-/62
	Empty	<b>33*41-/62*+</b>

# Infix to postfix Conversion Using Stack

**Example 2:** Suppose we are converting  $5 * (6 + 2) - (12/4)$  expression into postfix form using stack.



# Infix to postfix Conversion Using Stack

**Example 2:** Suppose we are converting  $5 * (6 + 2) - (12/4)$  expression into postfix form.

Following table shows the evaluation of Infix to Postfix:

	Symbol Scanned	Stack	Postfix Expression
1.	5		5
2.	*	*	5
3.	(	*, (	5, 6
4.	6	*, (	5, 6
5.	+	*, (, +	5, 6, 2
6.	2	*, (, +	5, 6, 2, +
7.	)	*	5, 6, 2, +, *
8.	-	-, (	5, 6, 2, +, *, 12
9.	(	-, (	5, 6, 2, +, *, 12
10.	12	-, (, 12	5, 6, 2, +, *, 12, 4
11.	/	-, (, 12, /	5, 6, 2, +, *, 12, 4, /
12.	4	-, (, 12, /	5, 6, 2, +, *, 12, 4, /, -
13.	)	-	
14.			

# Evaluation of a postfix Expression Using Stack

## Algorithm :

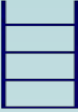


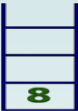





1. Scan the postfix expression from left to right.
2. If an operand is encountered, push it on stack.
3. If an operator '*op*' is encountered,
  - Pop two elements of stack, where A is the top element and B is the next top element.
  - Evaluate  $B \text{ } op \text{ } A$ .
  - Push the result on stack.
4. The evaluated value is equal to the value at the top of stack.

# Evaluation of a postfix Expression Using Stack

**Example :**

Infix :  $(5+3) * (8-2)$   
= 48

Postfix : 5,3,+,8,2,-,\*  
= 48

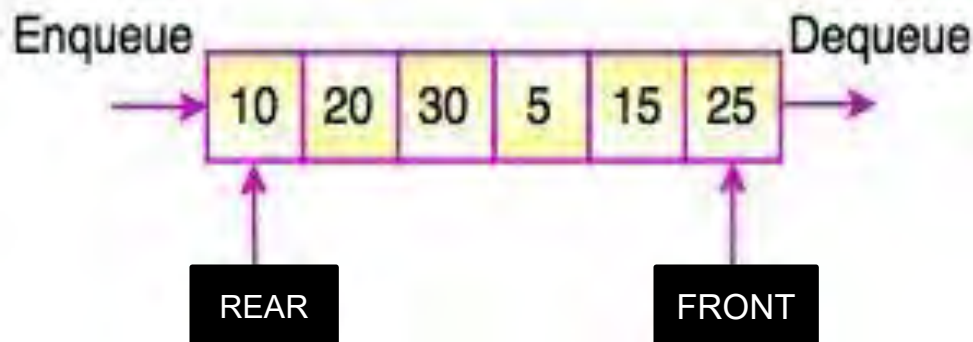
Reading Symbol	Stack Operations	Stack	Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)		Nothing
3	push(3)		Nothing
+	value1 = pop() value2 = pop() result = value2 + value1 push(result)		value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push( 8 ) <b>(5 + 3)</b>
8	push(8)		(5 + 3)
2	push(2)		(5 + 3)
-	value1 = pop() value2 = pop() result = value2 - value1 push(result)		value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push( 6 ) <b>(8 - 2)</b> (5 + 3), (8 - 2)
*	value1 = pop() value2 = pop() result = value2 * value1 push(result)		value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push( 48 ) <b>(6 * 8)</b> (5 + 3) * (8 - 2)
\$ End of Expression	result = pop()		Display (result) <b>48</b> As final result

# 5. QUEUE

- What is Queue ?
- Applications of Queue.
- Queue Representation.
- Types of Queue:
  - Simple Queue
  - Circular Queue
  - Priority Queue
  - Double Ended Queue
- Program to implement a queue using Array

# What is Queue ?

- Queue is a linear data structure where the first element is inserted from one end called **REAR** and deleted from the other end called as **FRONT**.
- **Front** points to the **beginning** of the queue and **Rear** points to the **end** of the queue.
- Queue follows the **FIFO (First - In - First Out)** structure.
- According to its FIFO structure, element inserted first will also be removed first.
- In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue), because queue is open at both its ends.
- The enqueue() and dequeue() are two important functions used in a queue.



# What is Queue ?

- A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.



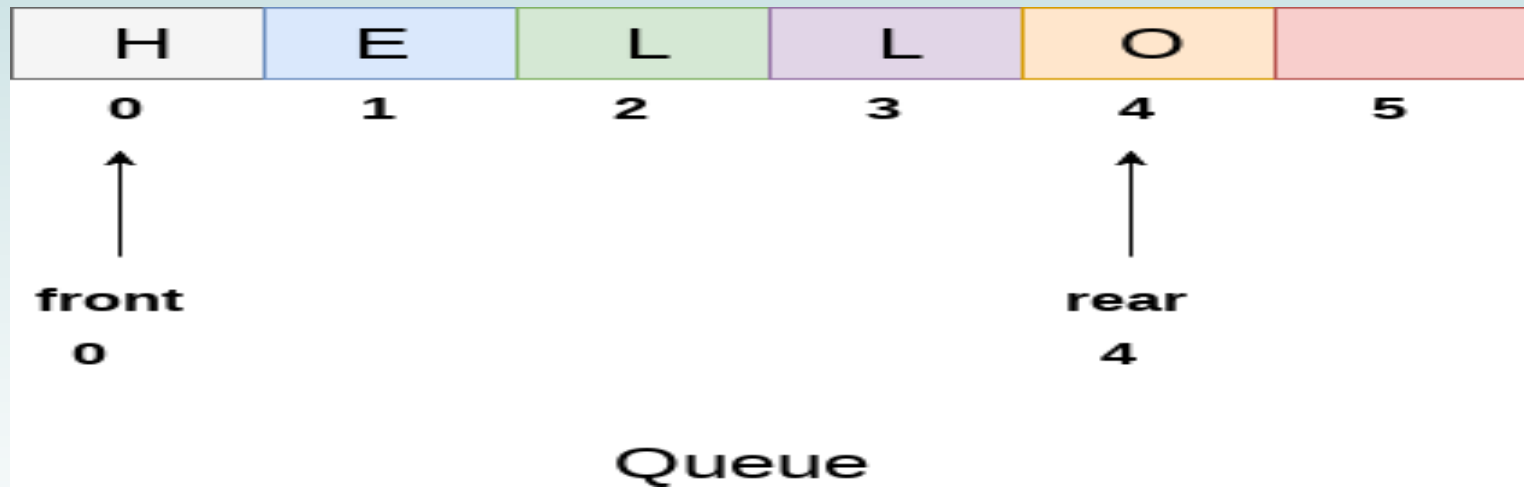
# Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

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

# Array Representation of Queue

- We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and rear is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.

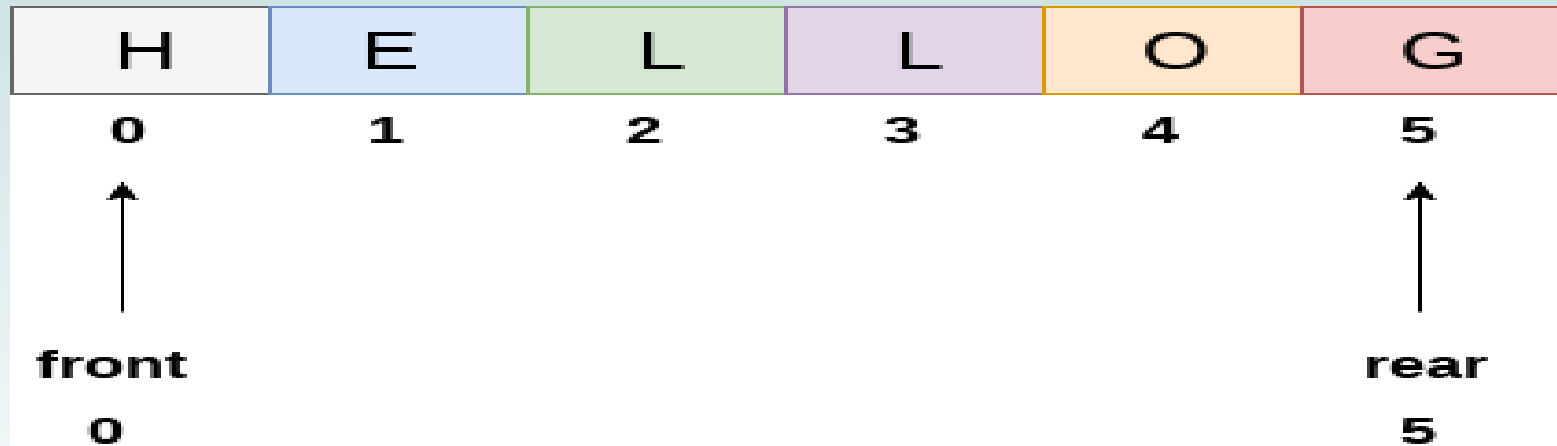


- The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains 0 .



# Array Representation of Queue

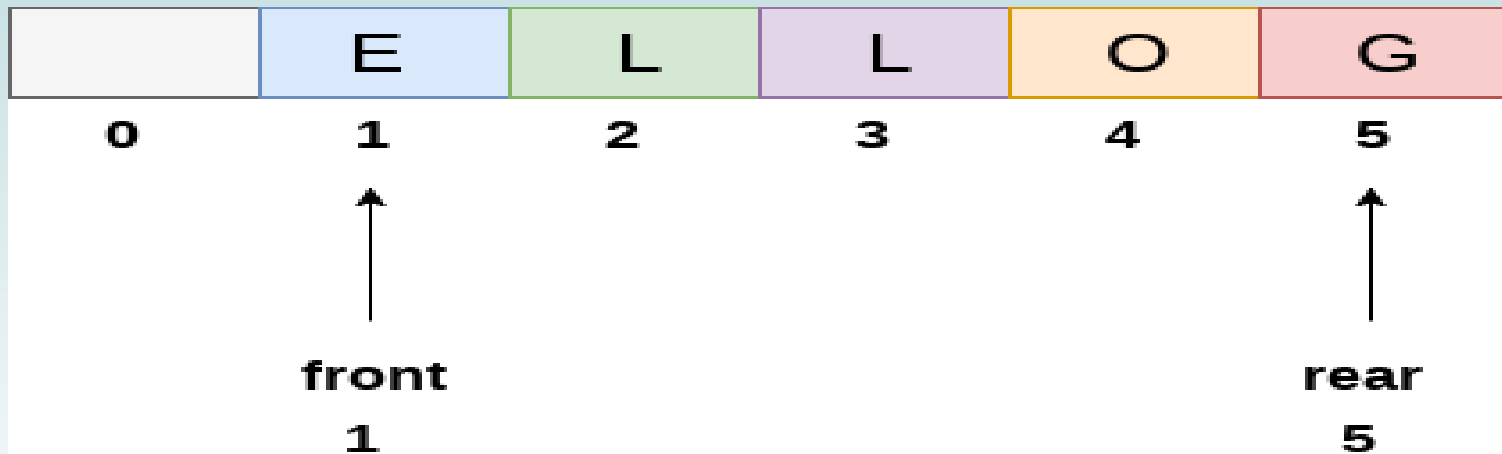
- However, the value of rear increases by one every time an insertion is performed in the queue. After inserting one more element into the queue shown in the previous figure, the queue will look something like following.
- The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

# Array Representation of Queue

- After deleting an element, the value of front will increase from 0 to 1. however, the queue will look something like following.



Queue after deleting an element

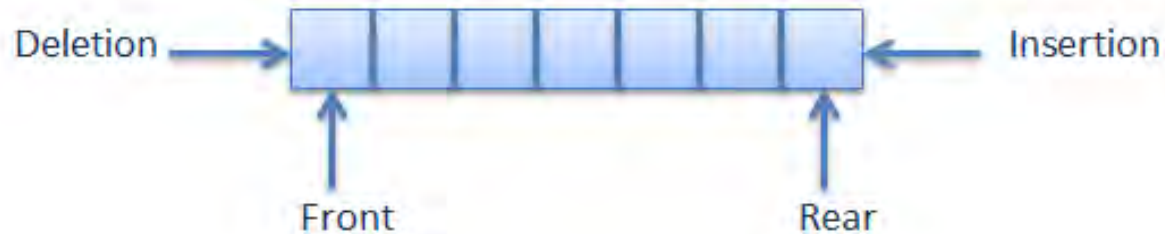
# Types of Queue

- **There are four types of Queue:**
  1. Simple Queue
  2. Circular Queue
  3. Priority Queue
  4. Double Ended Queue

# Types of Queue

## 1. Simple Queue :

- As is clear from the name itself, simple queue lets us perform the operations simply. i.e., the insertion and deletions are performed likewise. Insertion occurs at the rear (end) of the queue and deletions are performed at the front (beginning) of the queue list.
- Initially, the value of front and rear is -1 which represents an empty queue.



# Types of Queue

## 2. Circular Queue :

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as **Ring Buffer**.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

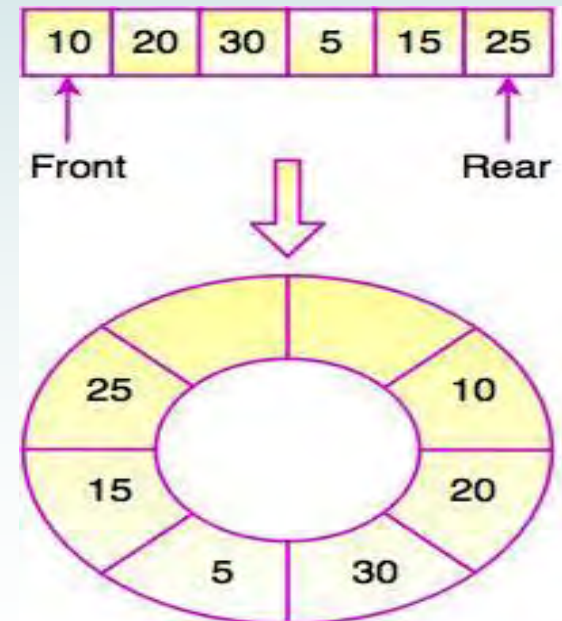
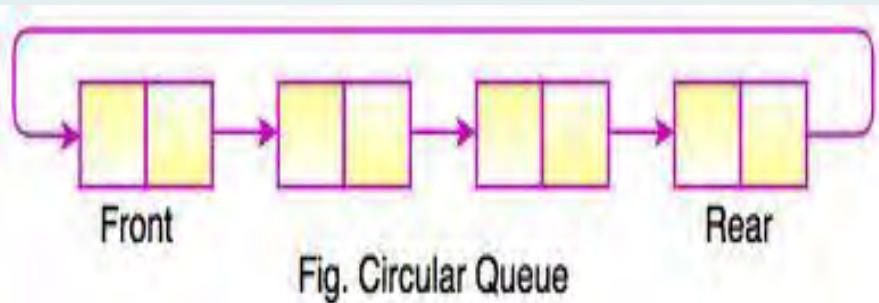
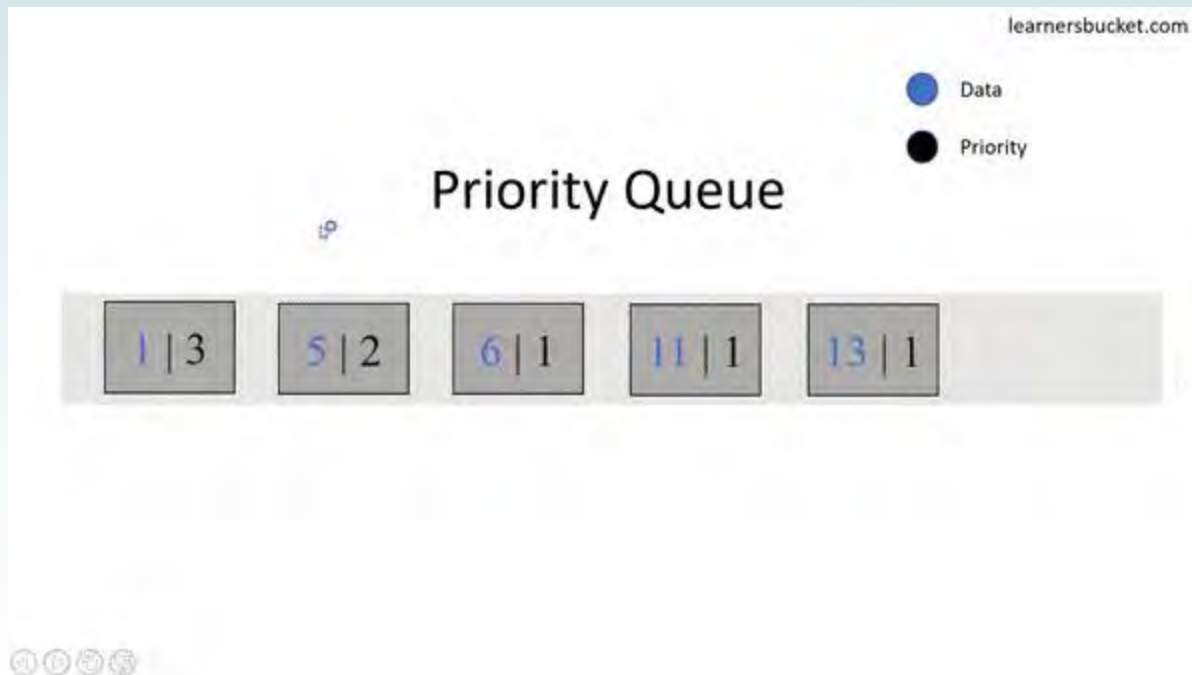


Fig. Circular Queue

# Types of Queue

## 3. Priority Queue :

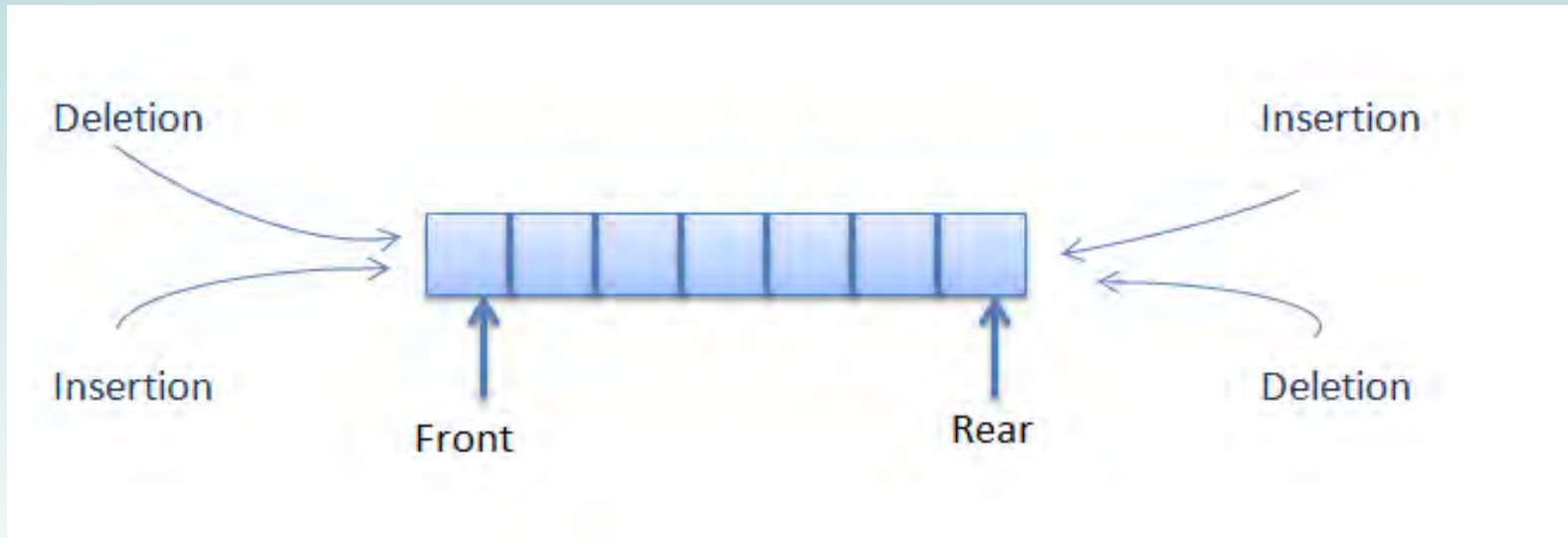
- Priority queue contains data items which have some preset priority. While removing an element from a priority queue, the data item with the highest priority is removed first.
- In a priority queue, insertion is performed in the order of arrival and deletion is performed based on the priority.



# Types of Queue

## 4. Double Ended Queue (Deque) :

- Double ended queue allows insertion and deletion from both the ends i.e. elements can be added or removed from rear as well as front end.

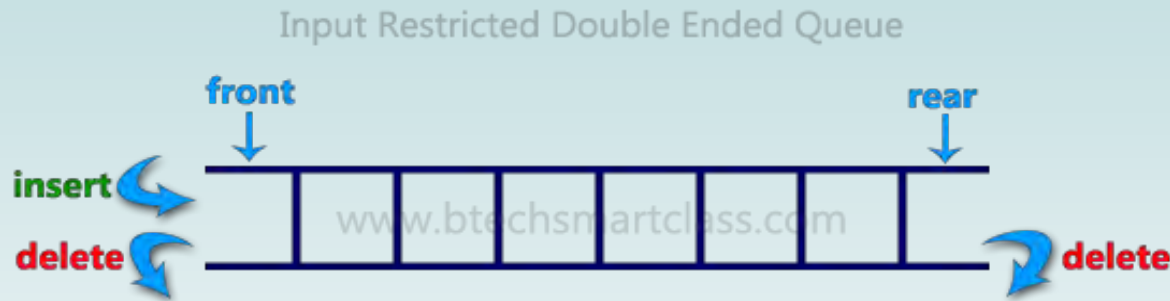


- There are two variations in Deque:
  - Input-Restricted Deque
  - Output-Restricted Deque.

# Types of Queue

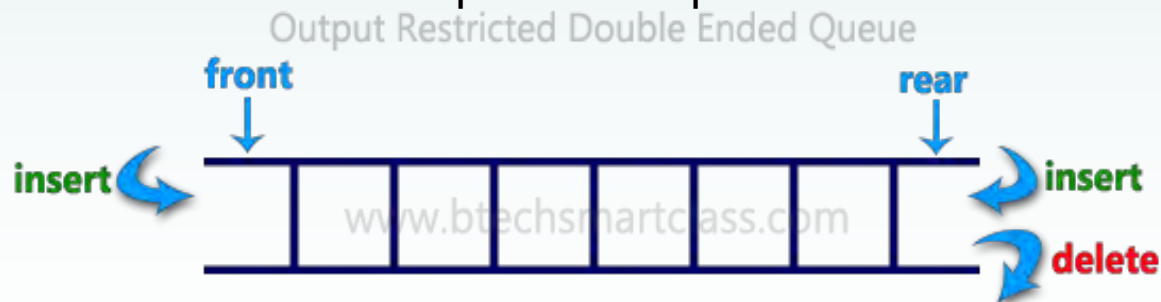
## Input Restricted Double Ended Queue

- In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



## Output Restricted Double Ended Queue

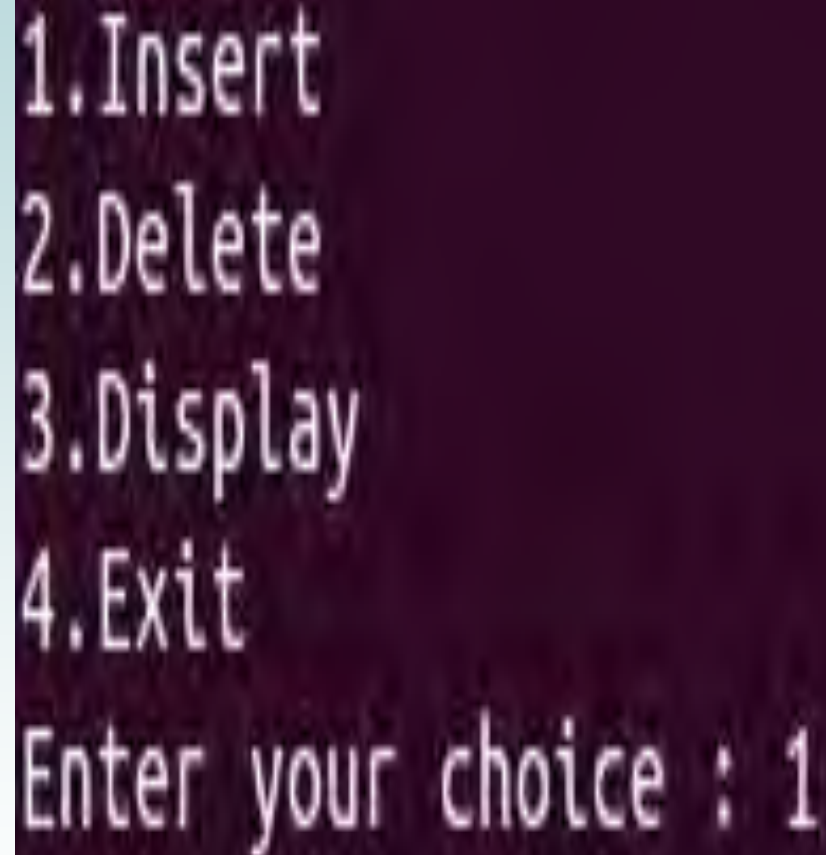
- In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.





# Program to implement a queue using Array

```
#include <stdio.h>
int queue_array[50];
int rear = - 1, front = - 1;
main()
{
    int choice;
    while (1)
    {
        printf("1.Insert \n");
        printf("2.Delete\n");
        printf("3.Display \n");
        printf("4.Exit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1: insert();
                    break;
            case 2: delete();
                    break;
            case 3: display();
                    break;
            case 4: exit(1);
                    break;
            default: printf("Inavlid choice \n");
        } /*End of switch*/
    } /*End of while*/
} /*End of main()*/
```



A screenshot of a terminal window with a dark purple background. The text is displayed in a light blue, monospaced font. It shows a menu with four options: 1.Insert, 2.Delete, 3.Display, and 4.Exit. Below the menu, the prompt 'Enter your choice : ' is followed by the user input '1'.

```
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
```

# Program to implement a queue using Array

```
insert()
{
    int add_item;
    if (rear == 49)
        printf("Queue Overflow \n");
    else
    {
        if (front == - 1)
            /*If queue is initially empty */
            front = 0;
        printf("Inset the element in queue : ");
        scanf("%d", &add_item);
        rear = rear + 1;
        queue_array[rear] = add_item;
    }
} /*End of insert()*/
```

```
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
Inset the element in queue : 10
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
Inset the element in queue : 20
```

# Program to implement a queue using Array

```
delete()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
        return ;
    }
    else
    {
        printf("Deleted Element is : %d\n",
queue_array[front]);
        front = front + 1;
    }
} /*End of delete() */
```

```
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 2
Deleted Element is : 10
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : █
```

# Program to implement a queue using Array

```
display()
{
    int i;
    if (front == - 1)
        printf("Queue is empty \n");
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");
    }
} /*End of display() */
```

```
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 3
Queue is :
10 20 30 40
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : |
```

# 6. Pointers

- Address in C.
- Pointer Syntax.
- Assigning Addresses to pointers.
- Changing the value pointed by pointers
- Working of pointers.

# Pointers in C programming

“Pointers are powerful features of C and C++ programming. Before we learn pointers, let's learn about addresses in C programming.”

# Address in C

- If you have a variable *var* in your program, *&var* will give you its address in the memory.
- We have used address numerous times while using the `scanf()` function:

`scanf (" %d" , &var) ;`

- Here, the value entered by the user is stored in the address of *var* variable. Let's take a working example.

## **Note :**

*& = ampersand*                      *'address of' operator*

*&a = address of a*              *printf("%d", &a)*

# Address in C

```
include <stdio.h>
int main()
{ int var = 5;
  printf("var: %d\n", var); // Notice the use of & before var
  printf("address of var: %d", &var);
  return 0;
}
```

Output:

var: 5

address of var: 266768

**Note:** You will probably get a different address when you run the above code.



# Pointer Syntax

- Pointers (pointer variables) are special variables that are used to store addresses rather than values.
- **Pointer Syntax**
  - Here is how we can declare pointers.  
`int *p;     // we have declared a pointer p of int type.`
  - You can also declare pointers in these ways.  
`Int *p1;   or   int * p2;`
  - Let's take another example of declaring pointers.  
`Int *p1, p2 ;   // Here, we have declared a pointer p1 and a normal variable p2.`

# Assigning addresses to Pointers

- Let's take an example.

```
int *pc, c ;  
c = 5 ;  
pc = &c ;
```

5

222

c (\*pc)

222

999

pc

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

## Note:

- In the above example, pc is a pointer, not \*pc. You cannot and should not do something like \*pc = &c ;
- & is 'address of' operator. For e.g &a means address of a.
- \* is 'value at address' operator. For e.g \*a means value at address stored in a.

# Changing Value Pointed by Pointers

- Let's take an example.

```
int* pc, c;  
c = 5;  
pc = &c;  
c = 1;  
printf("%d", c); // Output: 1  
printf("%d", *pc); // Output: 1
```

- We have assigned the address of c to the pc pointer.
- Then, we changed the value of c to 1. Since pc and the address of c is the same, \*pc gives us 1.

- Let's take another example.

```
int* pc, c;  
c = 5;  
pc = &c;  
*pc = 1;  
printf("%d", *pc); // Output: 1  
printf("%d", c); // Output: 1
```

- We have assigned the address of c to the pc pointer.
- Then, we changed \*pc to 1. Since pc and the address of c is the same, c will be equal to 1.

# Changing Value Pointed by Pointers

- Let's take one more example.

```
int* pc, c, d;  
c = 5;  
d = -15;  
pc = &c;  
printf("%d", *pc); // Output: 5  
pc = &d;  
printf("%d", *pc); // Output: -15
```

- Initially, the address of c is assigned to the pc pointer using `pc = &c;`. Since c is 5, `*pc` gives us 5.
- Then, the address of d is assigned to the pc pointer using `pc = &d;`. Since d is -15, `*pc` gives us -15.

# Working of Pointers

```
#include <stdio.h>
int main()
{
    int* pc, c;
    c = 22;
    printf("Address of c: %d\n", &c);
    printf("Value of c: %d\n\n", c); // 22
    pc = &c;
    printf("Address of pointer pc: %d\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); //
22
    c = 11;
    printf("Address of pointer pc: %d\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); //
11
    *pc = 2;
    printf("Address of c: %d\n", &c);
    printf("Value of c: %d\n\n", c); // 2
    return 0;
}
```

Address of c: 2686784

Value of c: 22

Address of pointer pc: 2686784

Content of pointer pc: 22

Address of pointer pc: 2686784

Content of pointer pc: 11

Address of c: 2686784

Value of c: 2

# Common mistakes when working with pointers

Suppose, you want pointer pc to point to the address of c, then

```
int c, *pc;
```

```
// pc is address but c is not
```

```
pc = c; // Error
```

```
// &c is address but *pc is not
```

```
*pc = &c; // Error
```

```
// both &c and pc are addresses
```

```
pc = &c;
```

```
// both c and *pc values
```

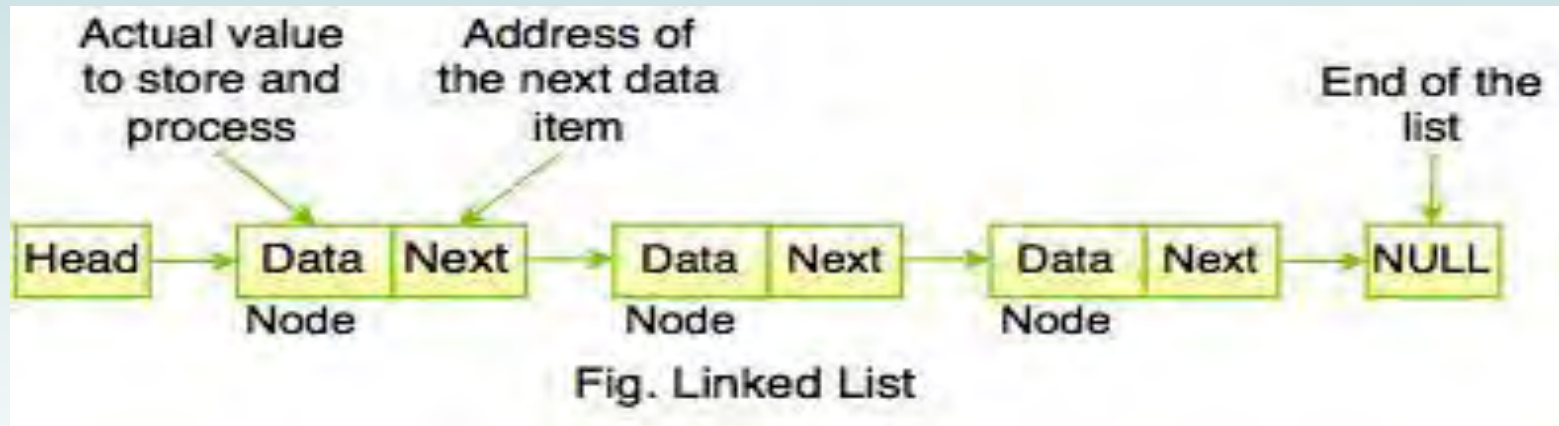
```
*pc = c;
```

# 7. Linked List

- Linked list: Definition.
- Advantages & disadvantages of linked list.
- Types of linked list.

# Linked List: Definition

- Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer.
- In linked list, each node consists of its own data and the address of the next node and forms a chain.



- The above figure shows the sequence of linked list which contains data items connected together via links. It can be visualized as a chain of nodes, where every node points to the next node.
- Linked list is a dynamic data structure.



# Advantages & Disadvantages

## **Advantages :**

- Linked list is dynamic in nature which allocates the memory when required.
- There is no need to define an initial size
- Insert and delete operation can be easily implemented in linked list.

## **Disadvantages :**

- Linked list has to access each node sequentially; no element can be accessed randomly.
- In linked list, the memory is wasted as pointer requires extra memory for storage.

# Types of Linked List

- **Following are the types of Linked List**
  1. Singly Linked List
  2. Doubly Linked List
  3. Circular Linked List
  4. Doubly Circular Linked List

# Singly Linked List

- Each node has a single link to another node is called Singly Linked List.
- Singly Linked List does not store any pointer to the previous node.
- Each node stores the contents of the node and a reference/address to the next node in the list.
- It has two successive nodes linked together in linear way and contains address of the next node to be followed.
- It has successor and predecessor. First node does not have predecessor while last node does not have successor. Last node have successor reference as NULL.

# Singly Linked List

- It has only single link for the next node.
- In this type of linked list, only forward sequential movement is possible, no direct access is allowed.

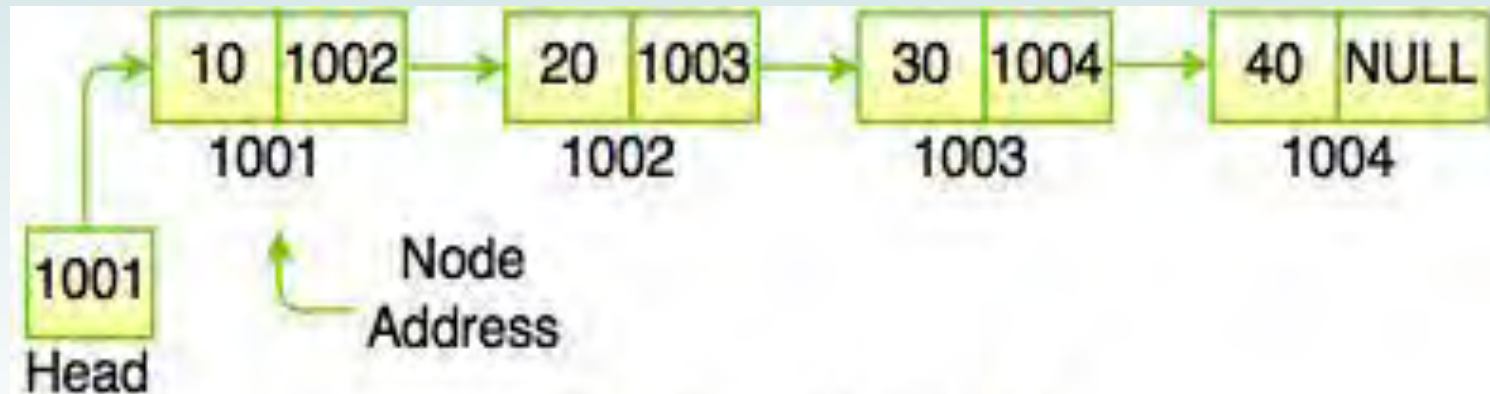


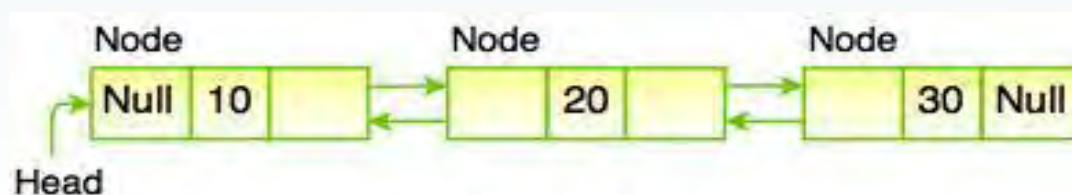
Fig. Singly Linked List

# Doubly Linked List

- Doubly linked list is a sequence of elements in which every node has link to its previous node and next node.
- Traversing can be done in both directions and displays the contents in the whole list.



- In the above figure, Link1 field stores the address of the previous node and Link2 field stores the address of the next node. The Data Item field stores the actual value of that node. If we insert a data into the linked list, it will be look like as follows:



# Doubly Linked List

## Advantages of Doubly Linked List

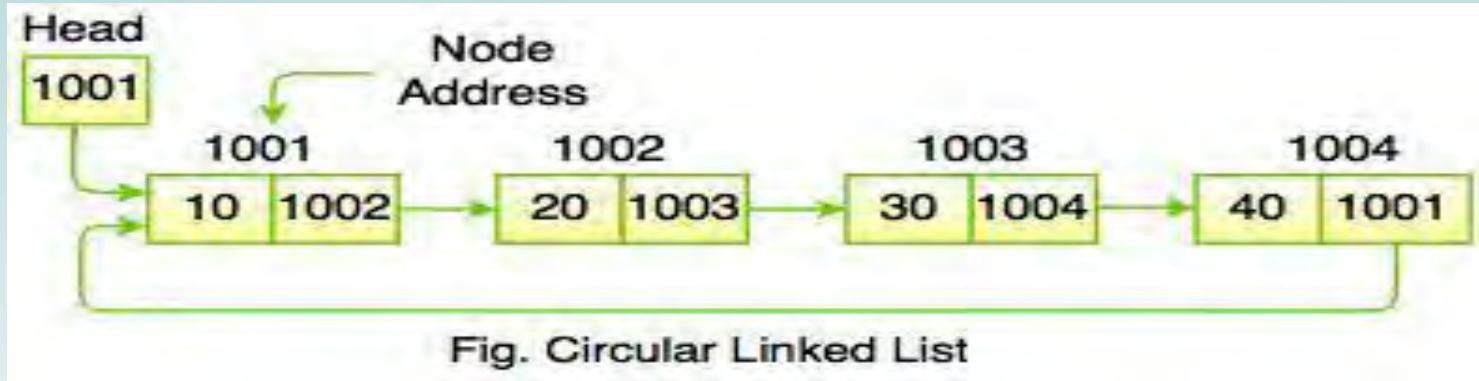
- Doubly linked list can be traversed in both forward and backward directions.
- To delete a node in singly linked list, the previous node is required, while in doubly linked list, we can get the previous node using previous pointer.
- It is very convenient than singly linked list. Doubly linked list maintains the links for bidirectional traversing.

## Disadvantages of Doubly Linked List

- In doubly linked list, each node requires extra space for previous pointer.
- All operations such as Insert, Delete, Traverse etc. require extra previous pointer to be maintained.

# Circular Linked List

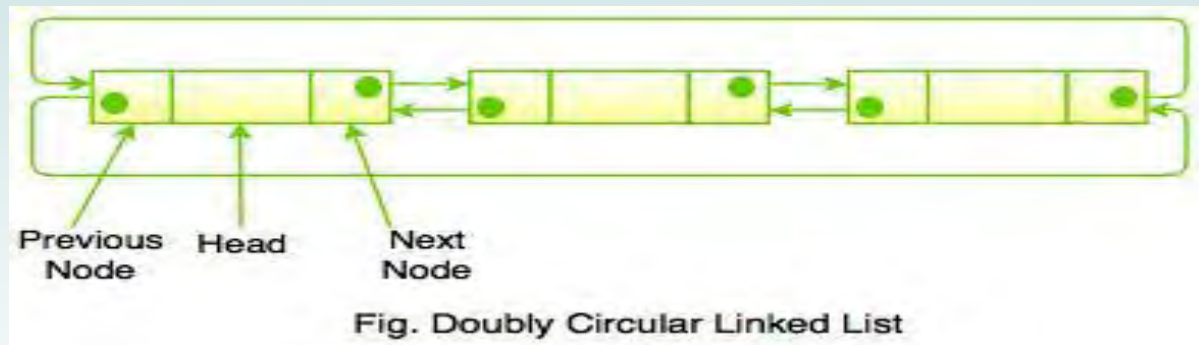
- Circular linked list is similar to singly linked list. The only difference is that in circular linked list, the last node points to the first node in the list.



- In the above figure we see that, each node points to its next node in the sequence but the last node points to the first node in the list. The previous node stores the address of the next node and the last node stores the address of the starting node.
- Circular linked list is used in personal computers, where multiple applications are running. The operating system provides a fixed time slot for all running applications and the running applications are kept in a circular linked list until all the applications are completed. This is a real life example of circular linked list.

# Circular Doubly Linked List

- A circular doubly linked list or a circular two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- The next field of the last node stores the address of the first node of the list.
- Similarly, the previous field of the first field stores the address of the last node.



- The previous link of the first node points to the last node and the next link of the last node points to the first node.
- In doubly circular linked list, each node contains two fields called links used to represent references to the previous and the next node in the sequence of nodes.



# 8. Tree Traversal

# Binary Tree Traversal

- Traversal is the process of visiting every node once.
- Visiting a node means doing some processing at that node, but when describing a traversal strategy, we need not concern ourselves with what that processing is.
- Three recursive techniques for binary tree traversal
- **In each technique, the left subtree is traversed recursively, the right subtree is traversed recursively, and the root is visited**
- What distinguishes the techniques from one another is the order of those 3 tasks

# Preorder, Inorder, Postorder

- In Preorder, the root is visited before (pre) the subtrees traversals
- In Inorder, the root is visited in-between left and right subtree traversal
- In Postorder, the root is visited after (post) the subtrees traversals

## Preorder Traversal:

1. Visit the root
2. Traverse left subtree in preorder
3. Traverse right subtree in preorder

## Inorder Traversal:

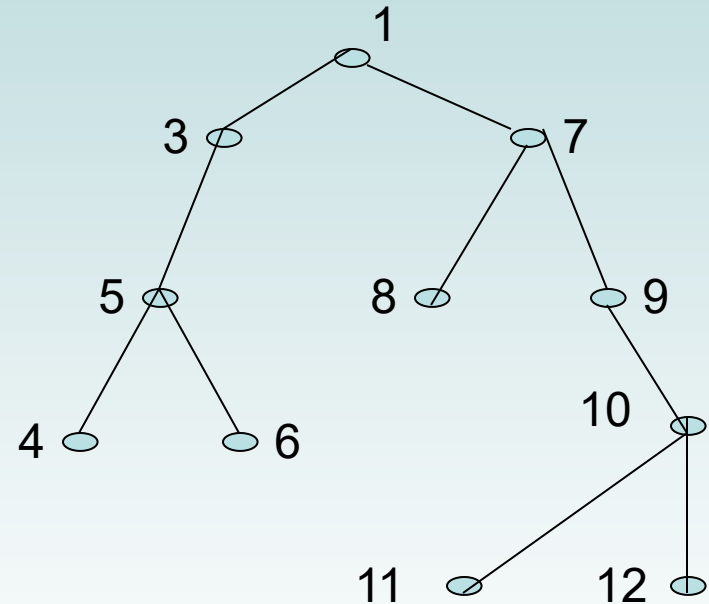
1. Traverse left subtree in inorder
2. Visit the root
3. Traverse right subtree in inorder

## Postorder Traversal:

1. Traverse left subtree in postorder
2. Traverse right subtree in postorder
3. Visit the root

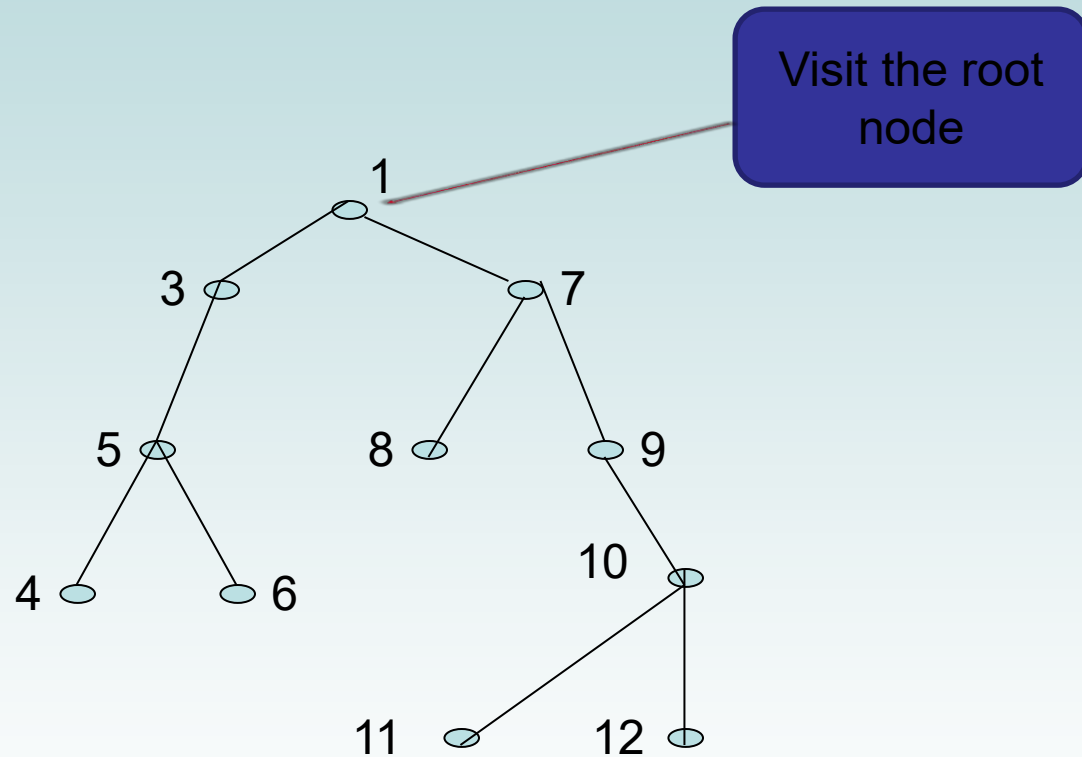
# Example of Traversal

- **Assume:** visiting a node is printing its label
- **Preorder:**  
1 3 5 4 6 7 8 9 10 11 12
- **Inorder:**  
4 5 6 3 1 8 7 9 11 10 12
- **Postorder:**  
4 6 5 3 8 11 12 10 9 7 1



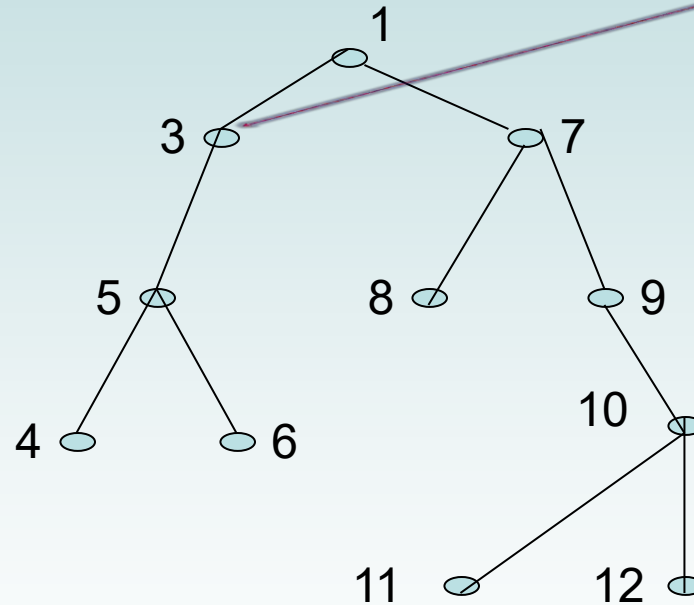
# Example of Traversal

- Preorder: 1,



# Example of Traversal

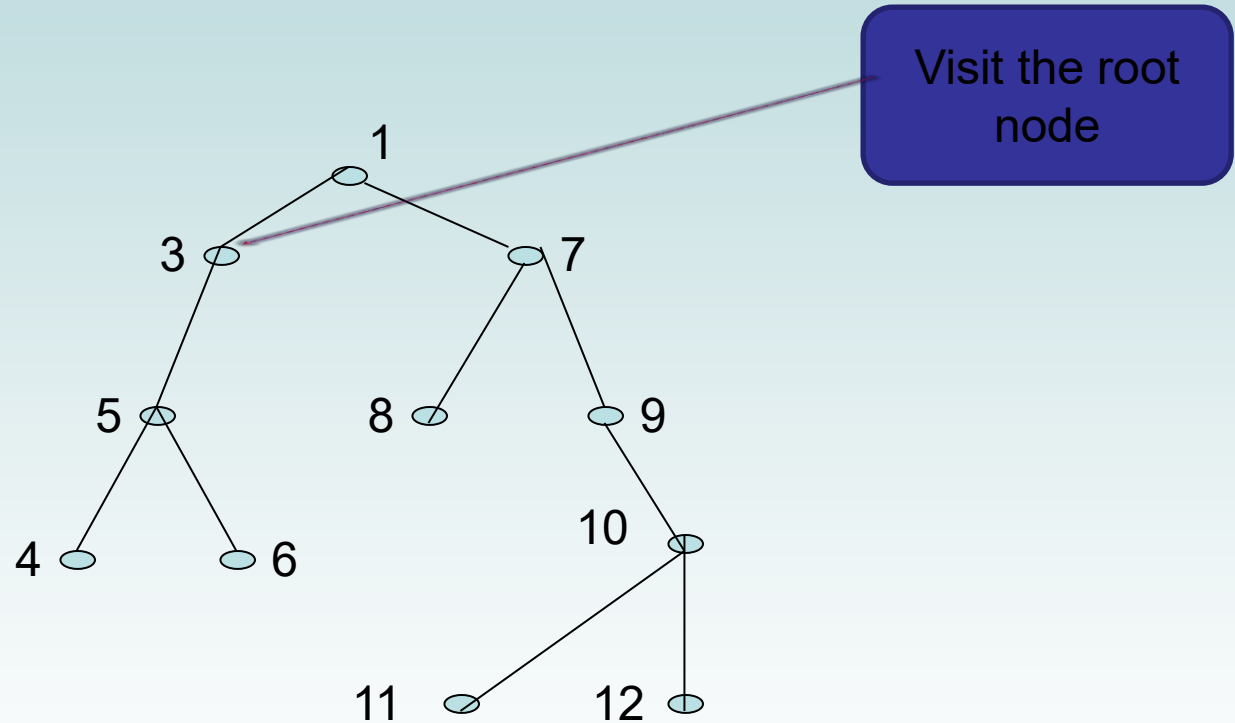
- Preorder: **1**,



Visit the left  
subtree in  
preorder

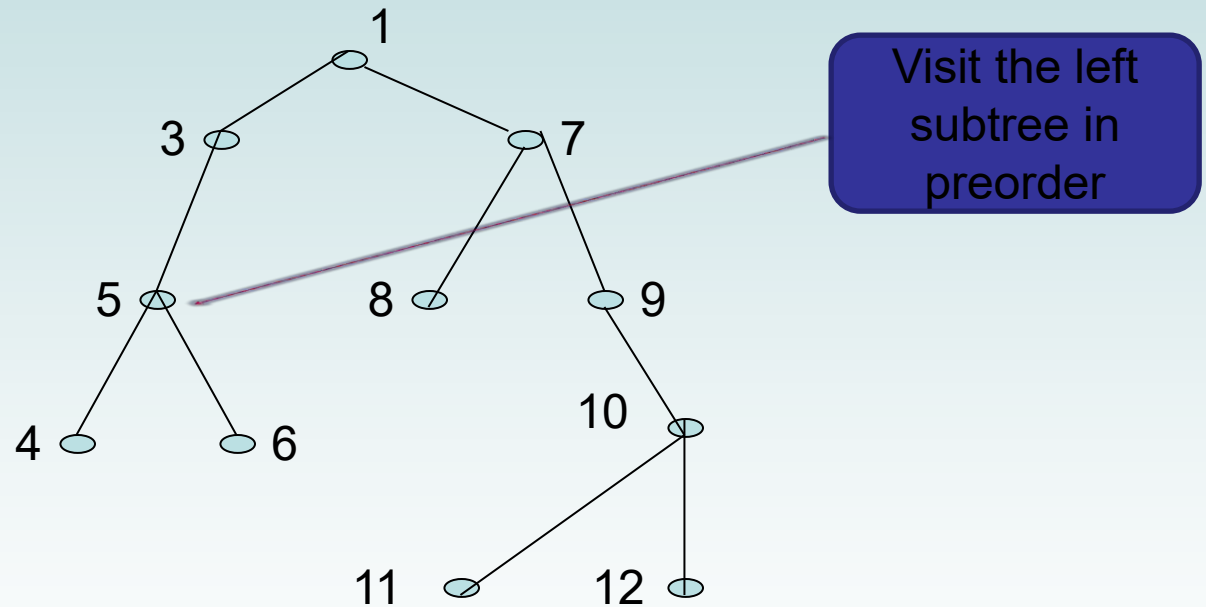
# Example of Traversal

- Preorder: 1, 3,



# Example of Traversal

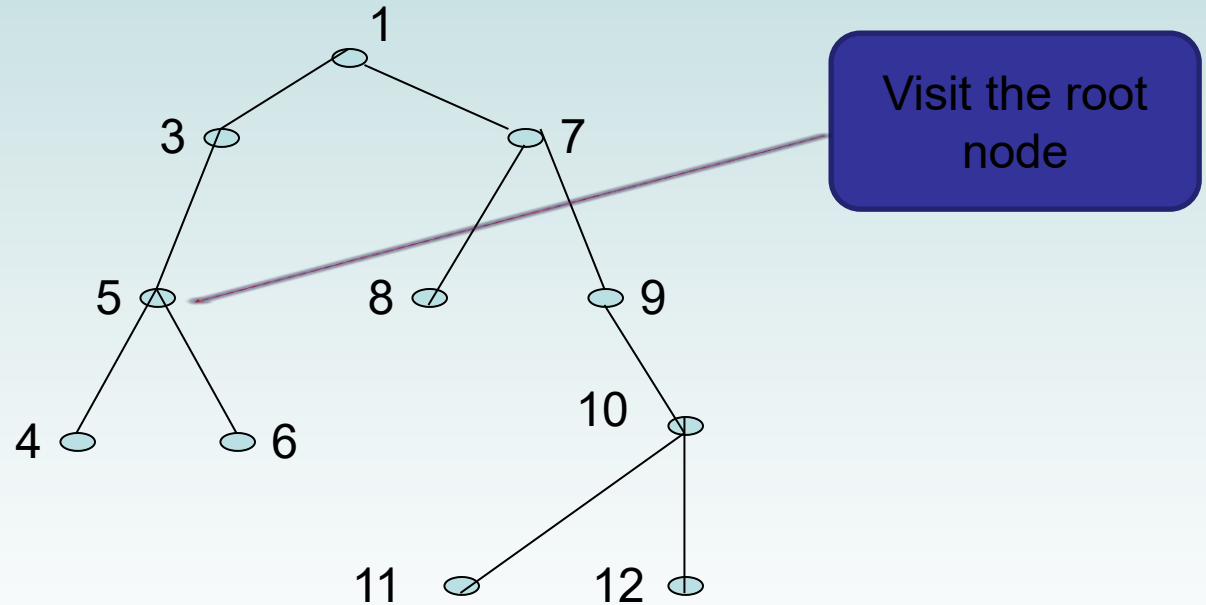
- Preorder: 1, 3,





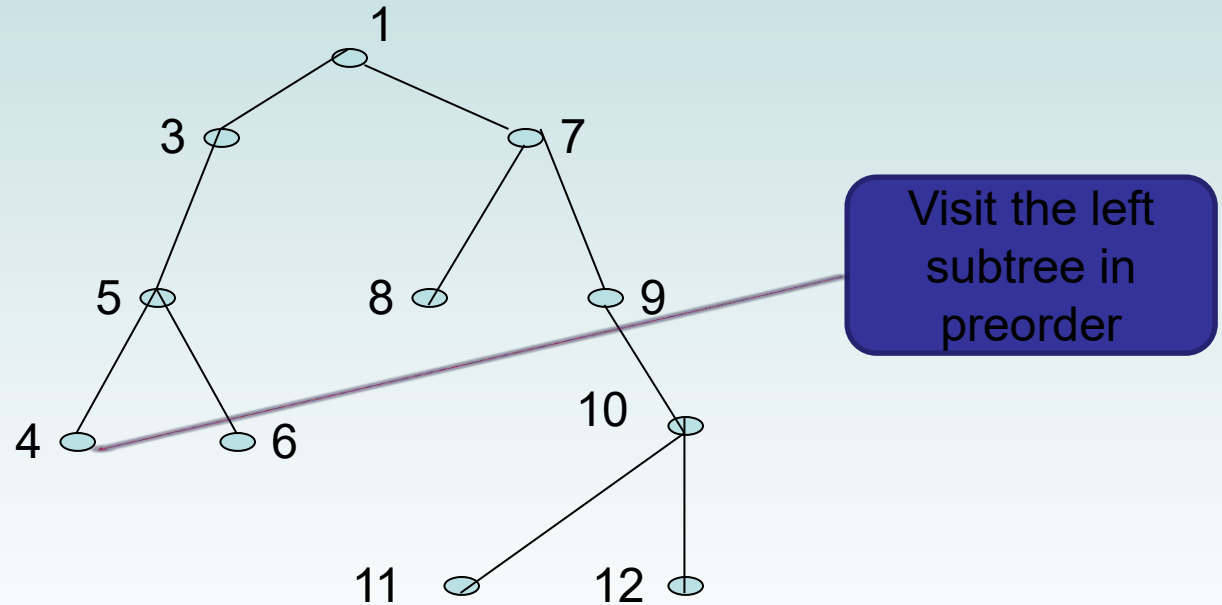
# Example of Traversal

- Preorder: 1, 3, 5



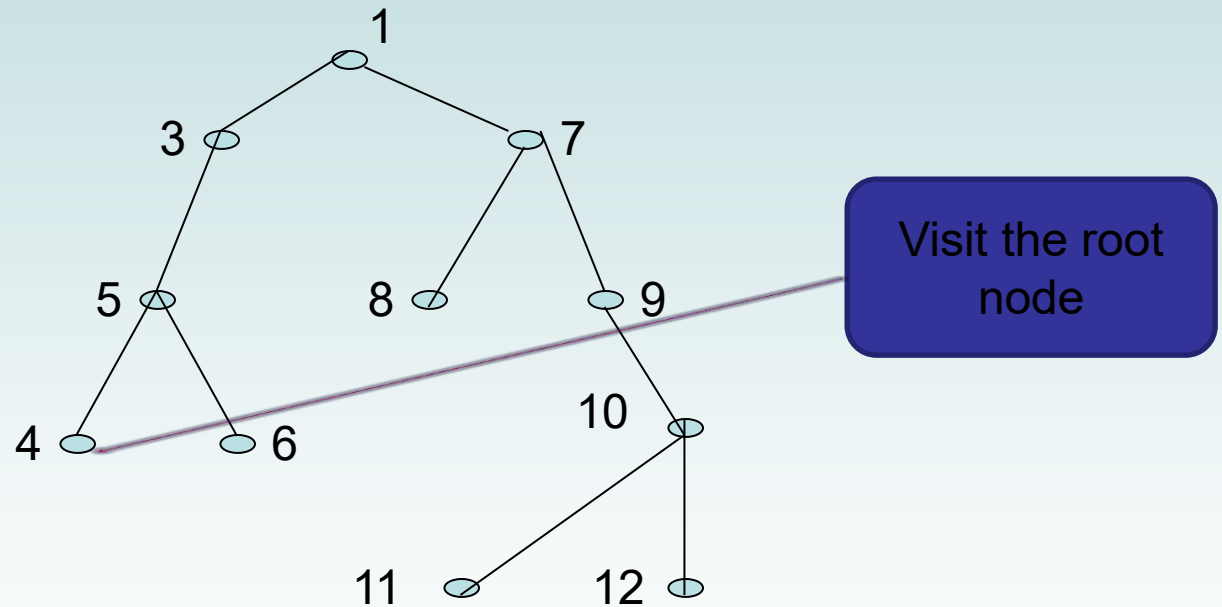
# Example of Traversal

- Preorder: 1, 3, 5



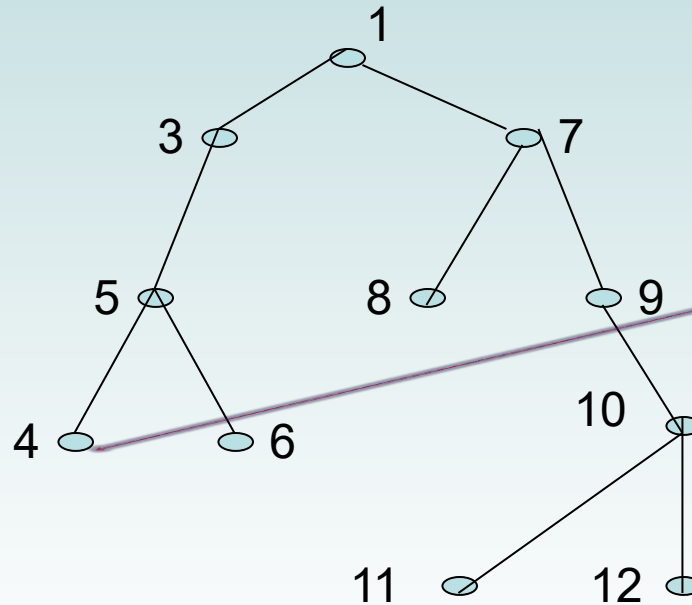
# Example of Traversal

- Preorder: 1, 3, 5, 4,



# Example of Traversal

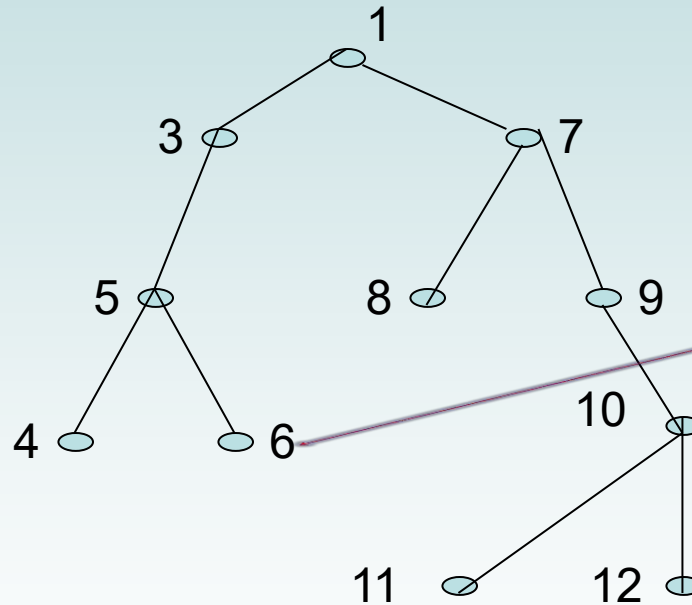
- Preorder: 1, 3, 5, 4,



There is no left or right subtree so preorder traversal is over at this node

# Example of Traversal

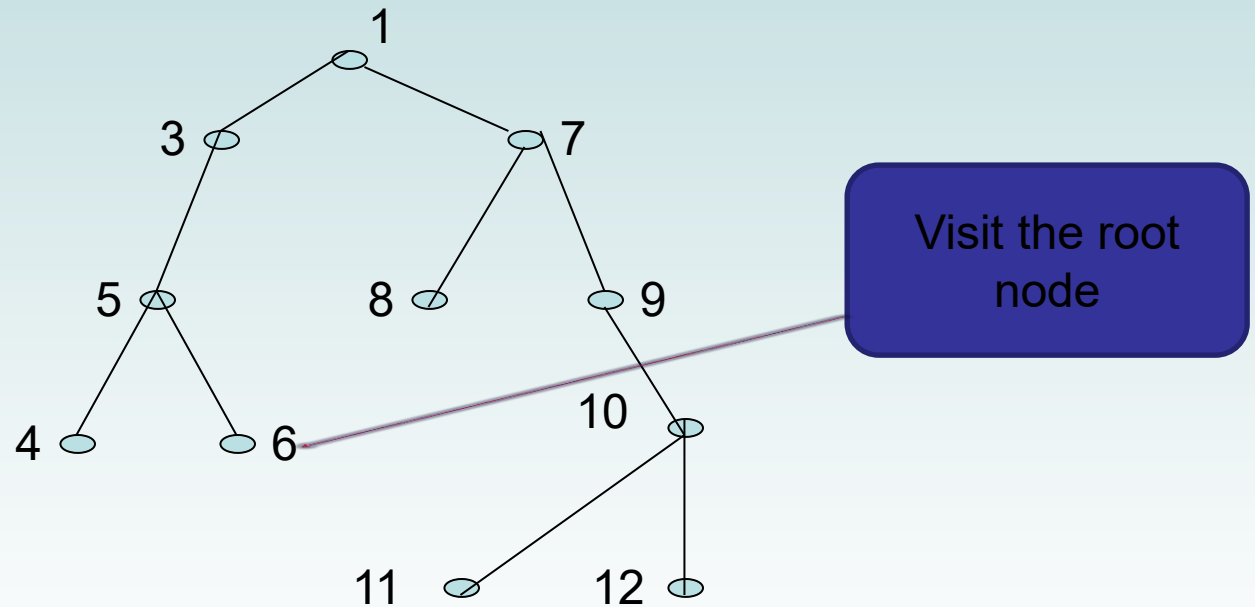
- Preorder: 1, 3, 5, 4,



Visit the right  
subtree in  
preorder

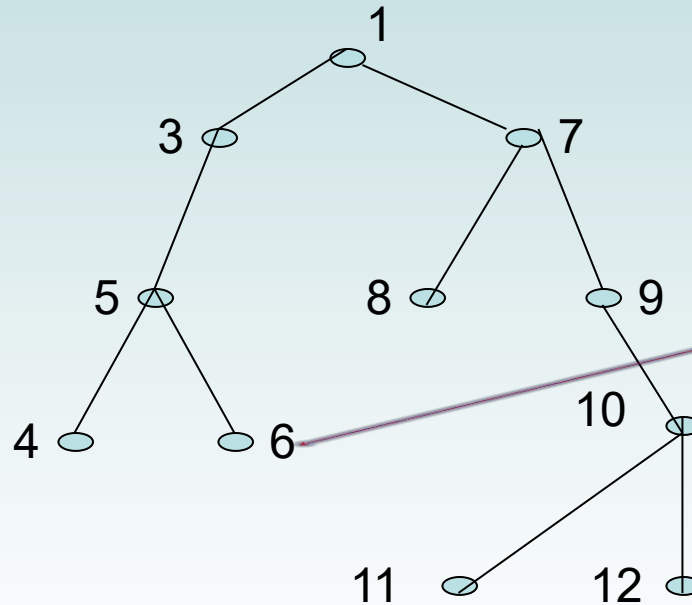
# Example of Traversal

- Preorder: 1, 3, 5, 4, **6**,



# Example of Traversal

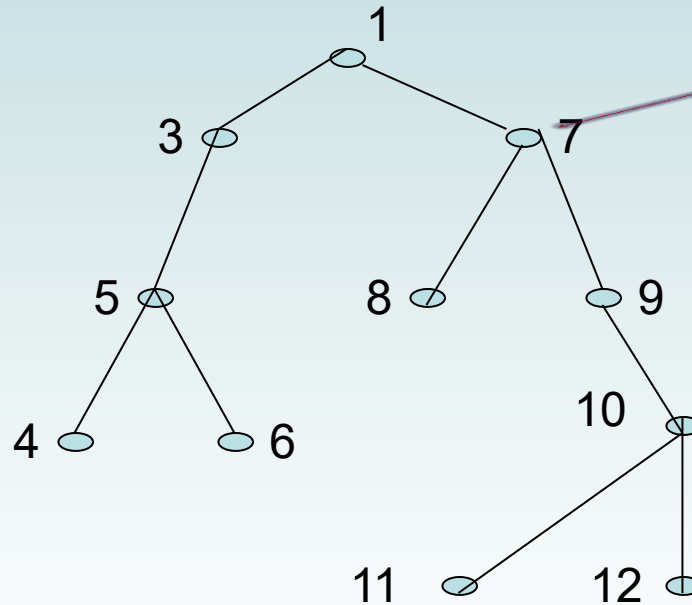
- Preorder: 1, 3, 5, 4, **6**,



There is no left or right subtree so preorder traversal is over at this node

# Example of Traversal

- Preorder: 1, 3, 5, 4, **6**,

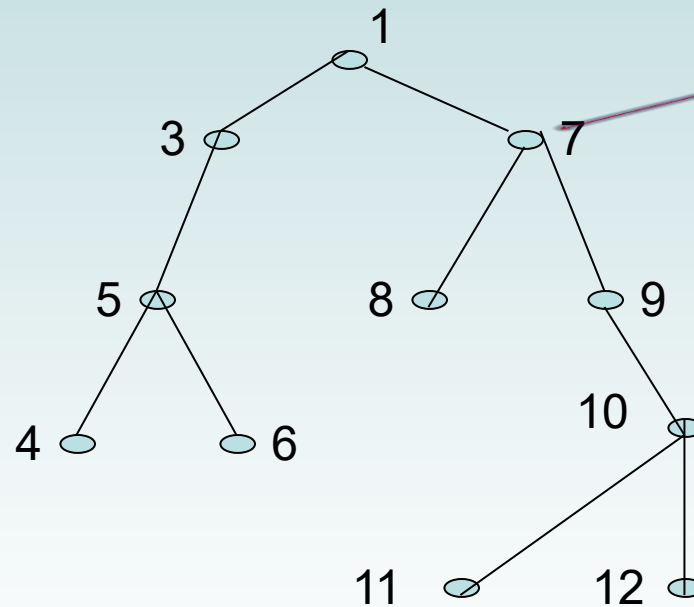


Visit the right  
subtree in  
preorder



# Example of Traversal

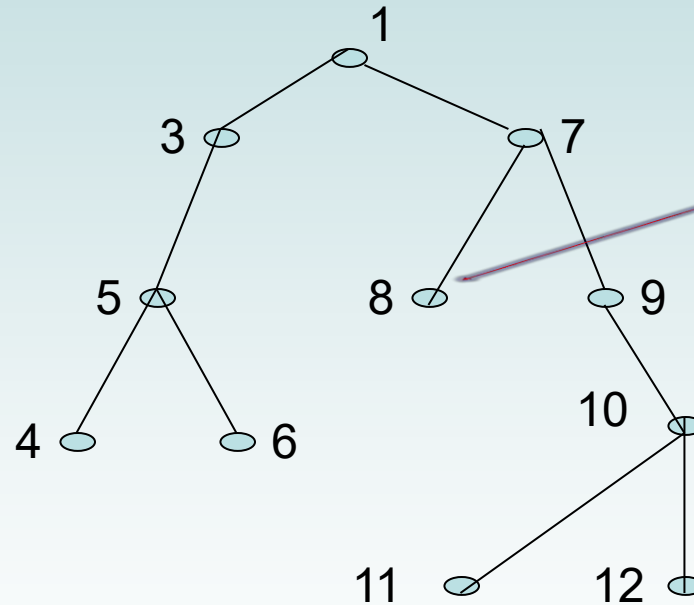
- Preorder: 1, 3, 5, 4, 6, 7,



Visit the root  
node

# Example of Traversal

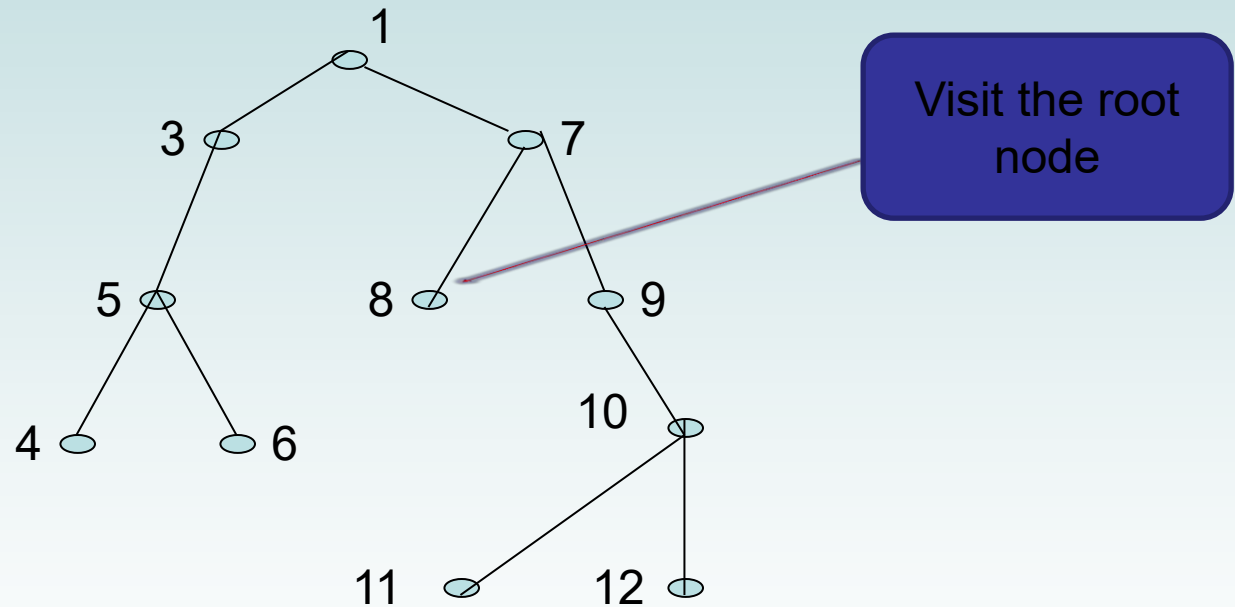
- Preorder: 1, 3, 5, 4, 6, 7,



Visit the left  
subtree in  
preorder

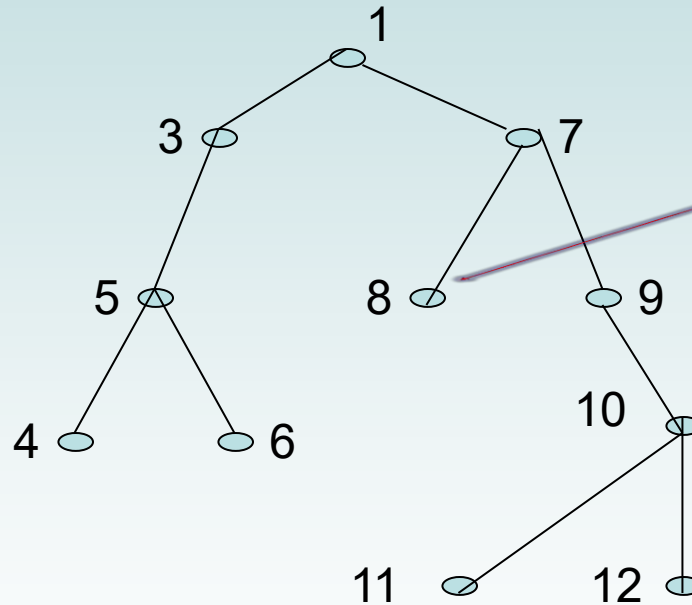
# Example of Traversal

- Preorder: 1, 3, 5, 4, 6, 7, 8,



# Example of Traversal

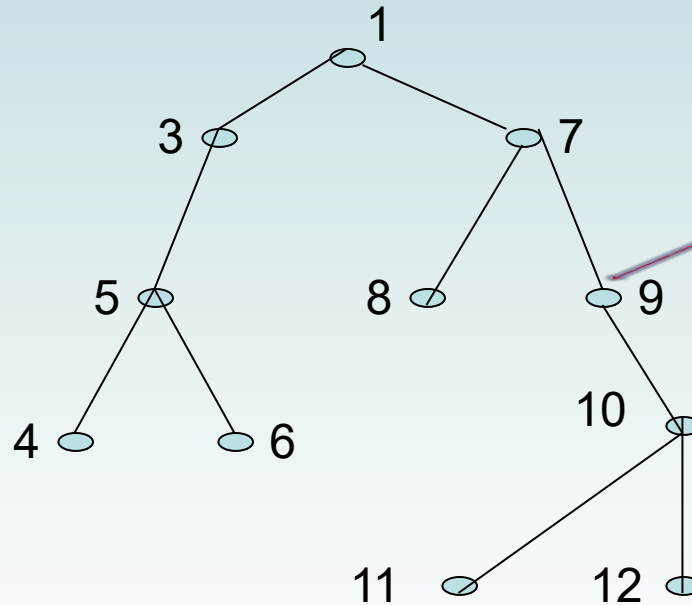
- Preorder: 1, 3, 5, 4, 6, 7, 8,



There is no left or right subtree so preorder traversal is over at this node

# Example of Traversal

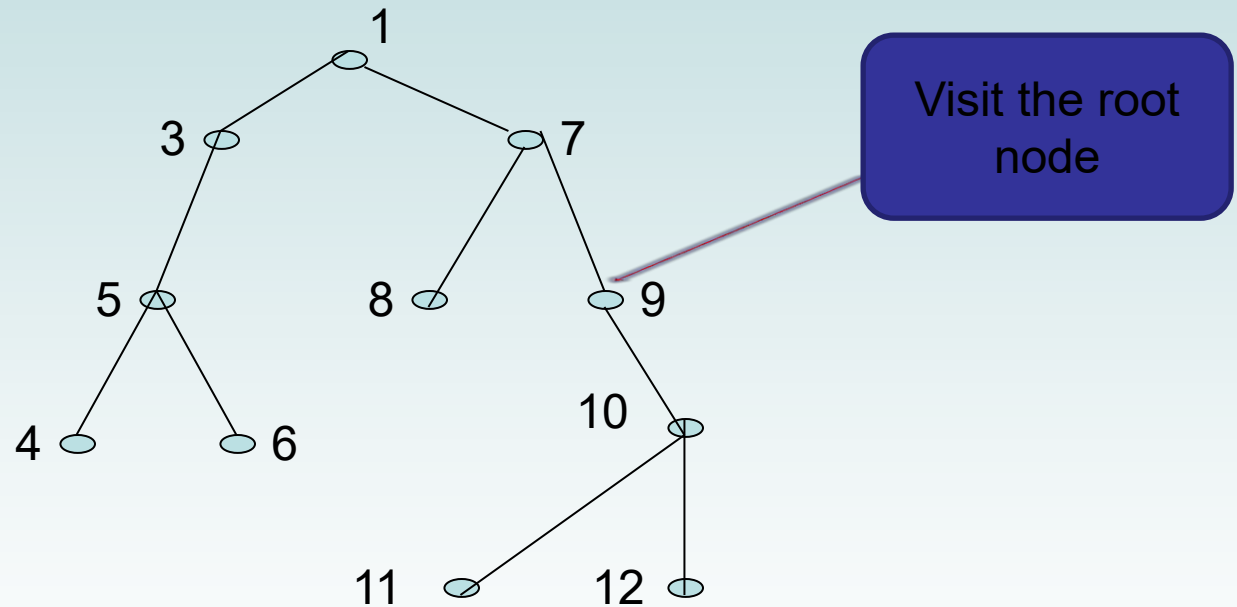
- Preorder: 1, 3, 5, 4, 6, 7, 8,



Visit the right  
subtree in  
preorder

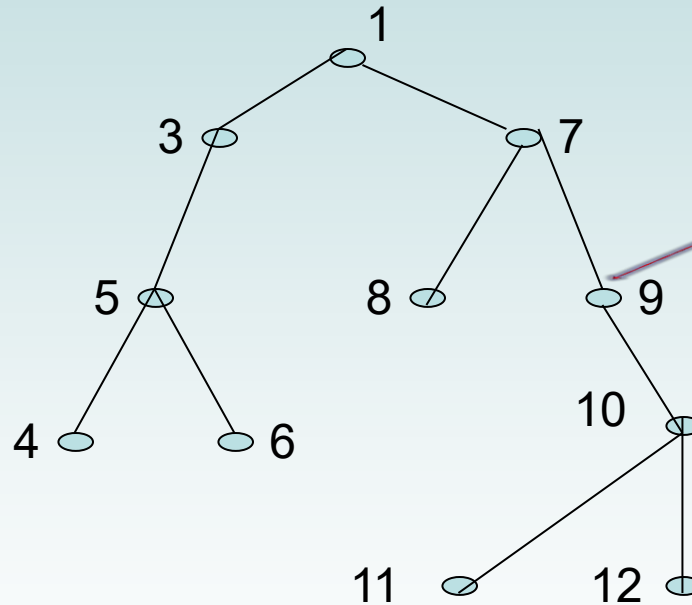
# Example of Traversal

- Preorder: 1, 3, 5, 4, 6, 7, 8, 9,



# Example of Traversal

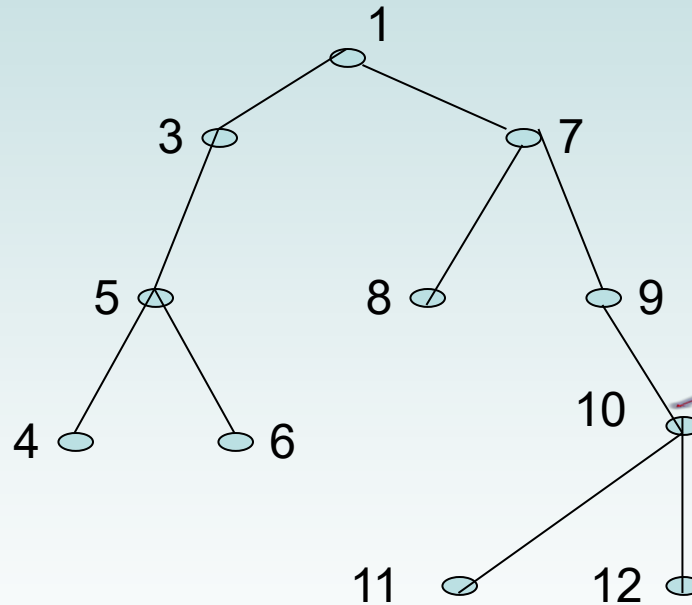
- Preorder: 1, 3, 5, 4, 6, 7, 8, 9,



There is no  
left subtree

# Example of Traversal

- Preorder: 1, 3, 5, 4, 6, 7, 8, 9, 10, 11, 12

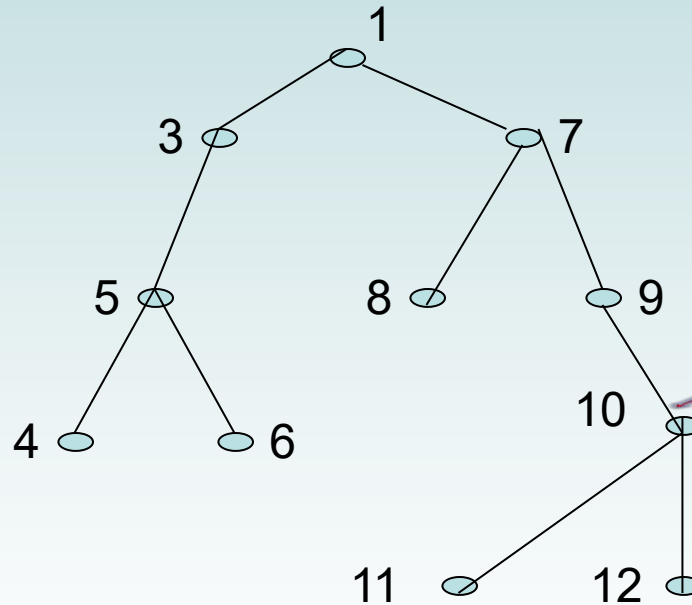


Visit the right  
subtree in  
preorder



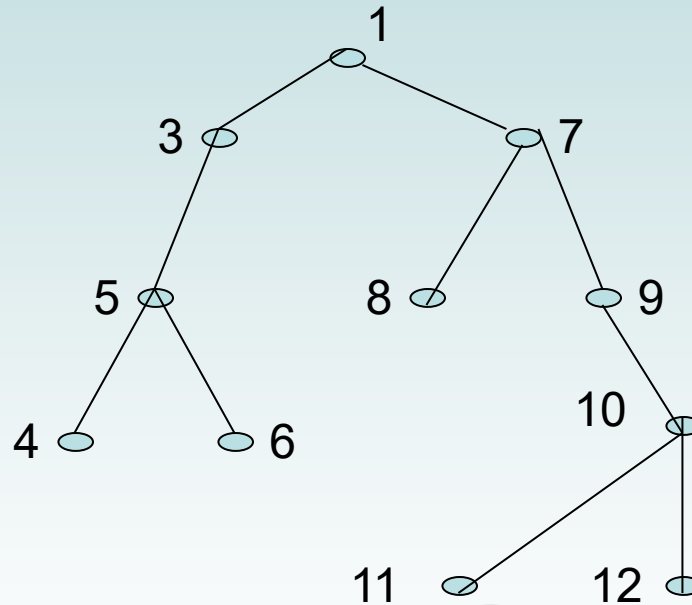
# Example of Traversal

- Preorder: 1, 3, 5, 4, 6, 7, 8, 9, **10**,



# Example of Traversal

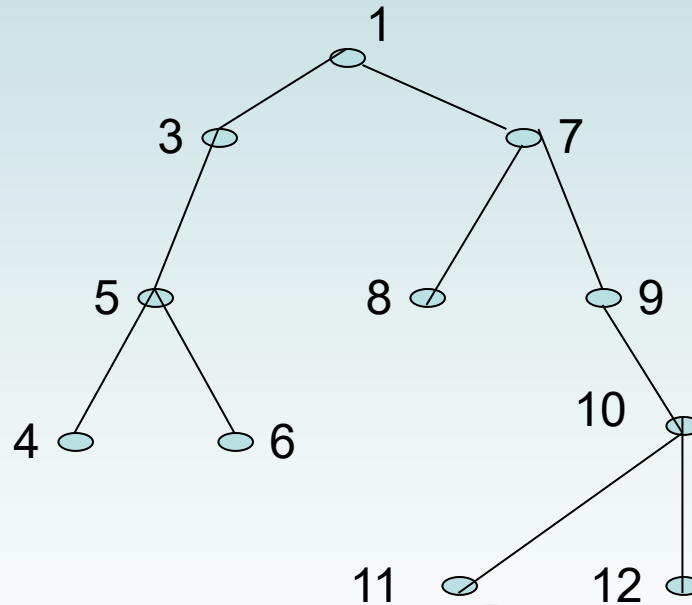
- Preorder: 1, 3, 5, 4, 6, 7, 8, 9, 10,



Visit the left  
subtree in  
preorder

# Example of Traversal

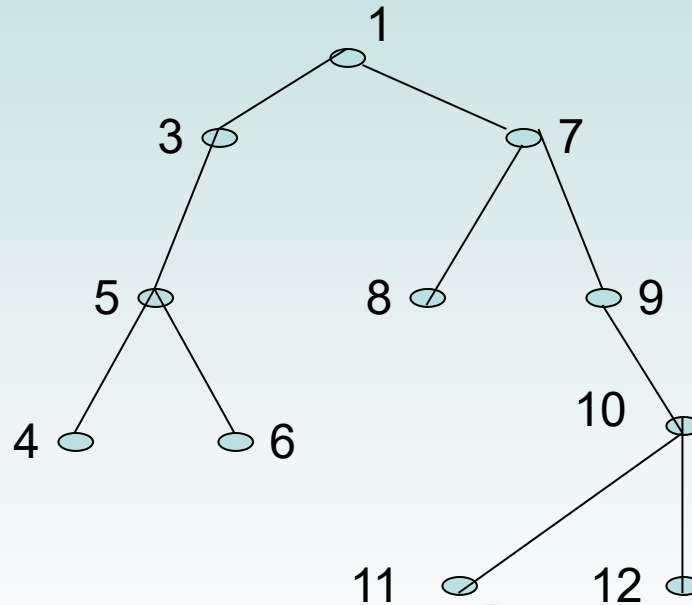
- Preorder: 1, 3, 5, 4, 6, 7, 8, 9, 10, **11**,



Visit the root  
node

# Example of Traversal

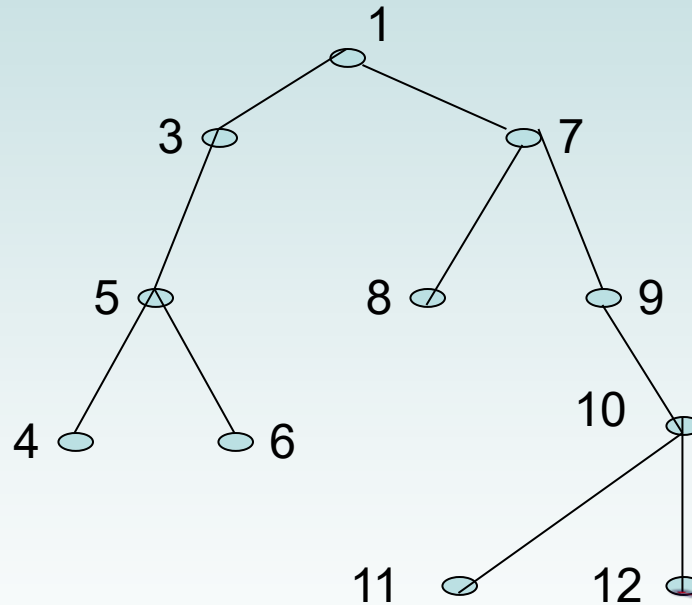
- Preorder: 1, 3, 5, 4, 6, 7, 8, 9, 10, **11**,



There is no left or right subtree so preorder traversal is over at this node

# Example of Traversal

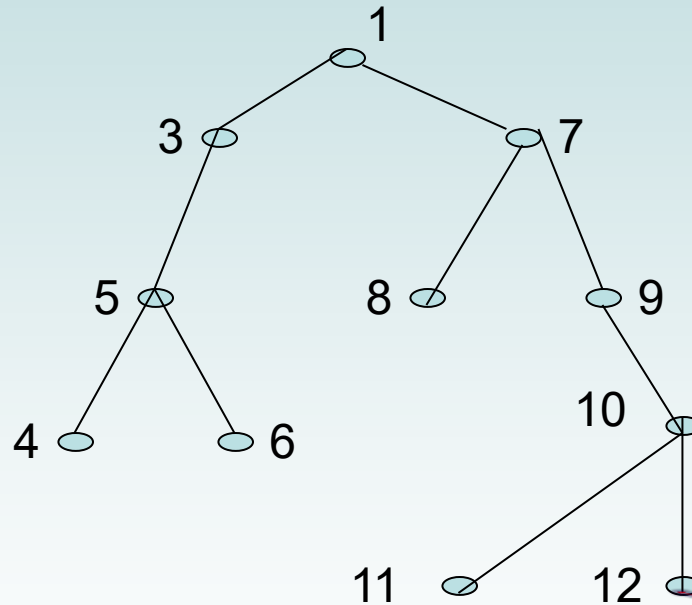
- Preorder: 1, 3, 5, 4, 6, 7, 8, 9, 10, **11**,



Visit the right  
subtree

# Example of Traversal

- Preorder: 1, 3, 5, 4, 6, 7, 8, 9, 10, 11, 12.



Visit the root  
node

# 9. Binary Search Tree (BST)

# Insertion Operation in BST

The insertion operation is performed as follows...

**Step 1** - Create a newNode with given value and set its **left** and **right** to **NULL**.

**Step 2** - Check whether tree is Empty.

**Step 3** - If the tree is **Empty**, then set **root** to **newNode**.

**Step 4** - If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).

**Step 5** - 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.

**Step 6**- Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).

**Step 7** - 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**.

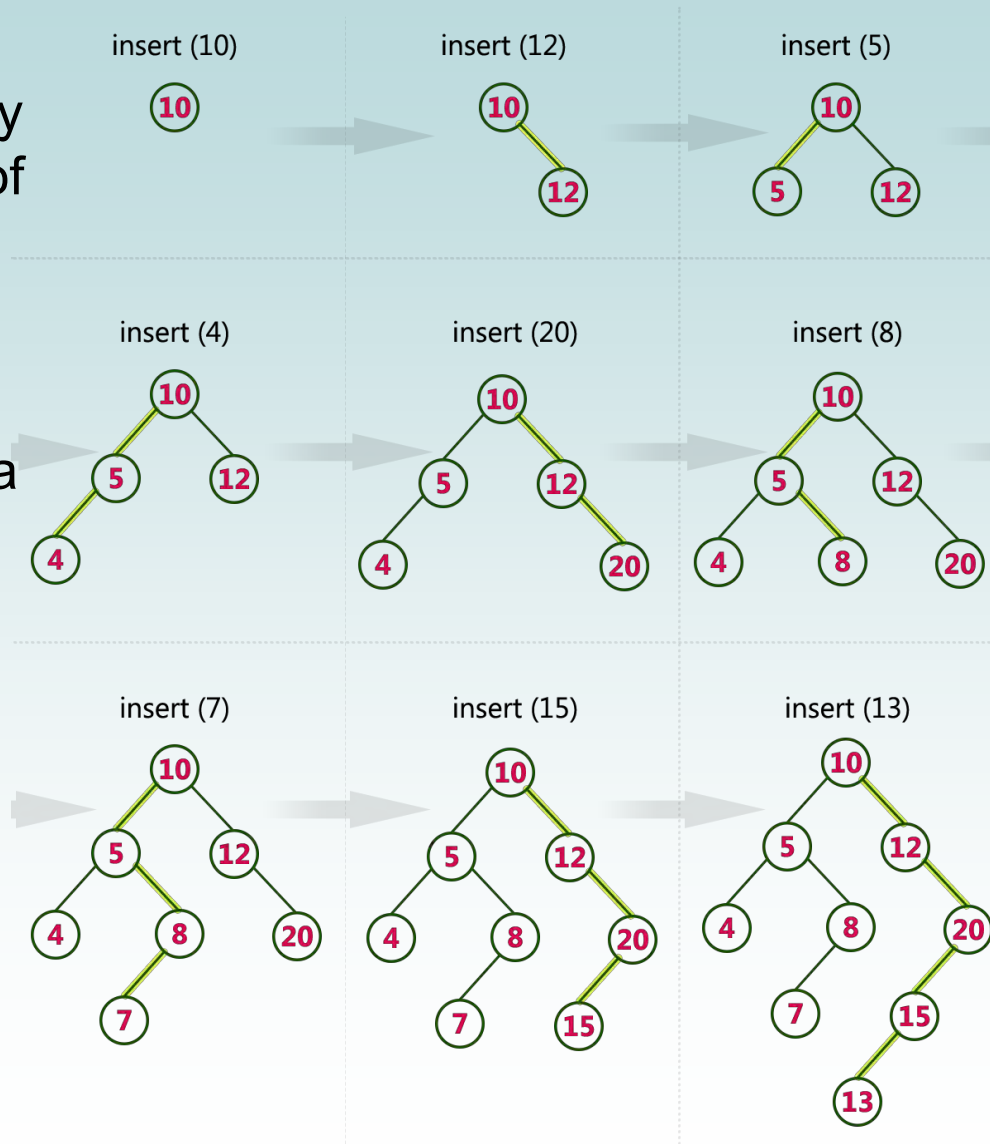


# Insertion Operation in BST

- \
- Construct a Binary Search Tree by inserting the following sequence of numbers...

**10, 12, 5, 4, 20, 8, 7, 15 and 13**

Above elements are inserted into a Binary Search Tree as follows...



# Deletion Operation in BST

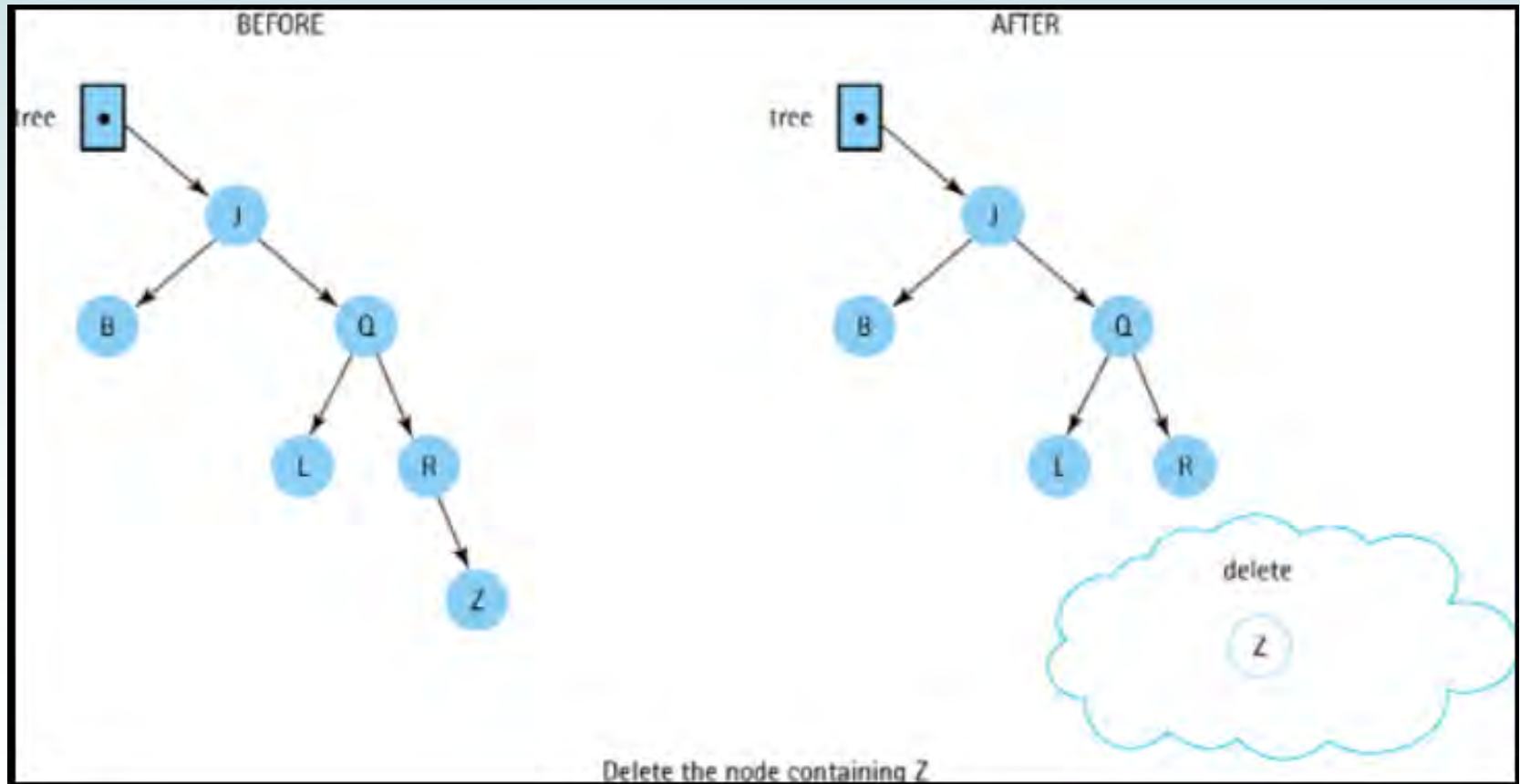
- In a binary search tree, the deletion operation is performed with  **$O(\log n)$**  time complexity. Deleting a node from Binary search tree includes following three cases...
  - **Case 1: Deleting a Leaf node (A node with no children)**
  - **Case 2: Deleting a node with one child**
  - **Case 3: Deleting a node with two children**

# Case 1: Deleting a leaf node

- We use the following steps to delete a leaf node from BST...

**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.



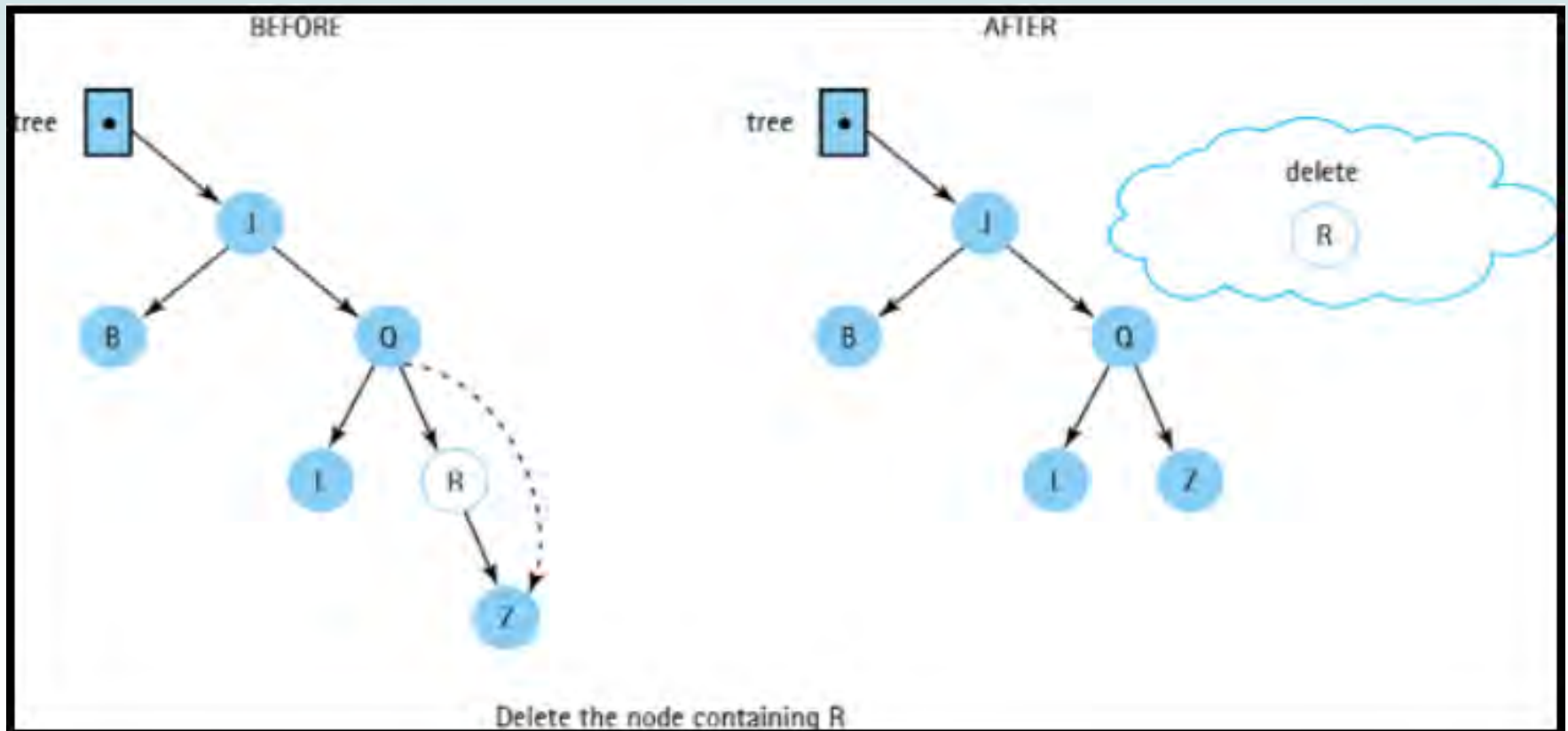
# Case 2: Deleting a node with one child

- We use the following steps to delete a node with one child from BST...

**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.



# Case 3: Deleting a node with two children

- We use the following steps to delete a node with two children from BST...

**Step 1** - Find the node to be deleted 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.

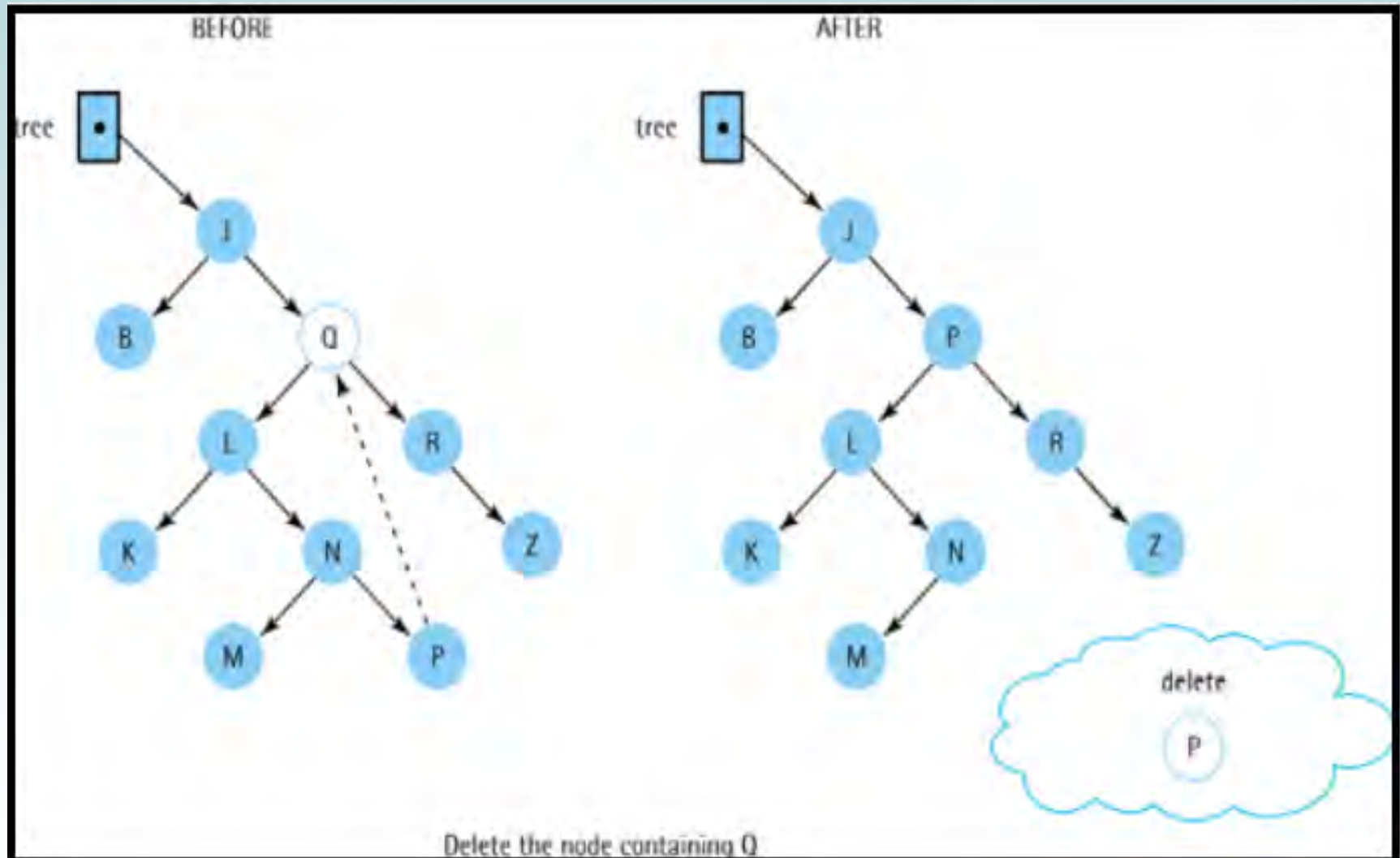
**Step 4** - Then check whether deleting node came to **case 1** or **case 2** or else goto step 2

**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.

## Case 3: Deleting a node with two children



# ASSIGNMENT

- Q 1. Explain all the cases of deletion operation in binary search tree.
- Q 2. Create a binary search tree from the following list of elements :
- 32, 14, 23, 40, 17, 48, 35, 45

THANK YOU