# DATA STRUCTURE

# Contents

# Data Structure

**Data Structure** can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science, for example: Operating System, Compiler Design, Artificial intelligence, Graphics etc.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

## Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminologies are used in data structures:

### Data

Data can be defined as an elementary value or the collection of values. ***for example***: student's **name** and its **id** are the data about the student.

### Group Items

Data items which have subordinate data items are called Group item, ***for example***: name of a student can have first name and the last name.

### Record

Record can be defined as the collection of various data items, ***for example***: if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File

A File is a collection of various records of one type of entity, ***for example***: if there are 60 students in the class, then there will be 20 records in the related file where each record contains the data about each student.

### Attribute and Entity

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

### Field

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

## Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

### Processor speed

To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

### Data Search

Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

### Multiple requests

If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process.

*In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.*

## Advantages of Data Structures

### Efficiency

Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

### Reusability

Data structures are reusable, 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.

### Abstraction

Data structure is specified by the ADT (Abstract Data Type) which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

## Classification of Data Structure

Based on the organizing method of data structure, data structures are divided into two types.

1) Linear Data Structures

2) Non - Linear Data Structures

## 1) Linear Data Structures

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

In other words, we can say that If a data structure organizes the data in sequential order, then that data structure is called a Linear Data Structure. Example:

- Arrays
- List (Linked List)
- Stack
- Queue

### Arrays

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional. The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

### Linked List

Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

### Stack

Stack is a linear list in which insertion and deletions are allowed only at one end, called top. A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

### Queue

Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

## 2) Non - Linear Data Structures

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in

sequential structure. Simply we can say that If a data structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure. Example:

- Tree
- Graph
- Dictionaries
- Heaps
- Tries, Etc.

### Trees

Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

### Graphs

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

## Operations on Data Structure

### 1. Traversing

Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

***Example***: If we need to calculate the average of the marks obtained by a student in 6 different subjects, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

### 2. Insertion

Insertion can be defined as the process of adding the elements to the data structure at any location. If the size of data structure is n then we can only insert n-1 data elements into it.

### 3. Deletion

The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then underflow occurs.

## 4. Searching

The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

## 5. Sorting

The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

## 6. Merging

When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

# Algorithm

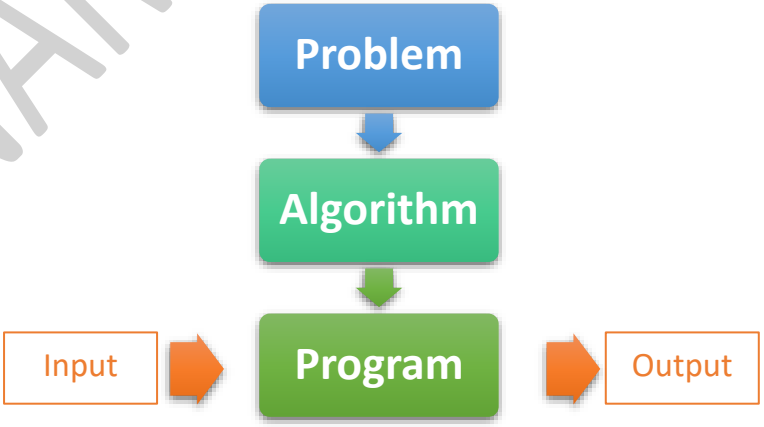
An algorithm is a step by step procedure to solve a problem. In normal language, the algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

---

*An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.*

---

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by a computer to produce a solution. Here, the program takes required data as input, processes data according to the program instructions and finally produces a result as shown in the following picture.




## Categories of Algorithm

The major categories of algorithms are given below:

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

## Performance of Algorithm

The performance of algorithm is measured on the basis of following properties:

- **Time complexity**: It is a way of representing the amount of time needed by a program to run to the completion.
- **Space complexity**: It is the amount of memory space required by an algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

Each algorithm must have:

- **Specification**: Description of the computational procedure.
- **Pre-conditions**: The condition(s) on input.
- **Body of the Algorithm**: A sequence of clear and unambiguous instructions.
- **Post-conditions**: The condition(s) on output.

***Example***: Design an algorithm to multiply the two numbers **x** and **y** and display the result in **z**.

**Step 1** START

**Step 2** declare three integers x, y & z

**Step 3** define values of x & y

**Step 4** multiply values of x & y

**Step 5** store the output of step 4 in z

**Step 6** print z

**Step 7** STOP

## Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

- **Input:** Every algorithm must take zero or more number of input values from external.
- **Output**: Every algorithm must produce an output as result.
- **Definiteness**: Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation).

- **Finiteness**: For all different cases, the algorithm must produce result within a finite number of steps.
- **Effectiveness**: Every instruction must be basic enough to be carried out and it also must be feasible.

## Performance Analysis of an algorithm

If we want to go from city "A" to city "B", there can be many ways of doing this. ***We can go by flight, by bus, by train and also by bicycle***. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick

> ***Performance of an algorithm is a process of making evaluative judgement about algorithms.***

the one which is best suitable for our requirements. The formal definition is as follows:

It can also be defined as follows:

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

We compare algorithms with each other which are solving the same problem, to select the

> ***Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.***

best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.

***Generally, the performance of an algorithm depends on the following elements:***

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.

Based on this information, performance analysis of an algorithm can also be defined as follows:

***Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.***

Performance analysis of an algorithm is performed by using the following measures:

1) Space Complexity
2) Time Complexity

## 1) Space Complexity

Space required to complete the task of that algorithm. It includes program space and data space. When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes:

- To store program instructions.
- To store constant values.
- To store variable values.
- And for few other things like function calls, jumping statements etc.

Space complexity of an algorithm can be defined as follows:

***Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.***

***Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows:***

**Instruction Space:** It is the amount of memory used to store compiled version of instructions.

**Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.

> ***Note -*** *When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack.* ***That means we calculate only the memory required to store Variables, Constants, Structures, etc.***

**Data Space:** It is the amount of memory used to store all the variables and constants.

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following:

- 2 bytes to store Integer value.
- 4 bytes to store Floating Point value.
- 1 byte to store Character value.
- 6 (OR) 8 bytes to store double value.

Consider the following piece of code:

**Example 1**

```
int square(int a)
{
      return a*a;
}
```

In the above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.

That means, totally it requires 4 bytes of memory to complete its execution. And these 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be ***Constant***

*If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.*

***Space Complexity*.**

Consider the following piece of code:

**Example 2**

```
int sum(int A[ ], int n)
{
      int sum = 0, i;
      for(i=0; i<n; i++)
            sum=sum+A[i];
      return sum;
}
```

**In the above piece of code, it requires:**

- 'n*2' bytes of memory to store array variable 'a[ ]'
- 2 bytes of memory for integer parameter 'n'
- 4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each)
- 2 bytes of memory for return value.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount of memory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be ***Linear Space Complexity***.

> *If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be **Linear Space Complexity.***

## 2) Time Complexity

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity. The time complexity

> *The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.*

of an algorithm can be defined as follows:

Generally, the running time of an algorithm depends upon the following:

- Whether it is running on Single processor machine or Multi processor machine.
- Whether it is a 32-bit machine or 64-bit machine.
- Read and Write speed of the machine.
- The amount of time required by an algorithm to perform Arithmetic operations, logical operations, return value and assignment operations etc.
- Input data

***Note*** *- When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.*

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system. To solve this problem, we must assume a model machine with a specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration:

- It is a Single processor machine

- It is a 32-bit Operating System machine
- It performs sequential execution
- It requires 1 unit of time for Arithmetic and Logical operations
- It requires 1 unit of time for Assignment and Return value
- It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above-defined model machine. Consider the following piece of code:

**Example 1**

```
int sum(int a, int b)
{
      return a+b;
}
```

In the above sample code, it requires 1 unit of time to calculate **a+b** and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of **a** and **b**. That means for all input values, it requires the same amount of time i.e. 2 units.

*If any program requires a fixed amount of time for all input values then its time complexity is said to be* ***Constant Time Complexity.***

Consider the following piece of code:

**Example 2**

```
int sum(int A[], int n)
{
int sum=0,i;
      for(i=0;i<n;i++)
            sum=sum+A[i];
      return sum;
}
```

For the above code, time complexity can be calculated as follows:

- Cost is the amount of computer time required for a single operation in each line.
- Repetition is the amount of computer time required by each operation for all its repetitions.
- Total is the amount of computer time required by each operation to execute.
- So above code requires **'4n+4'** Units of computer time to complete the task. Here the exact time is not fixed. And it changes based on the n value. If we increase the n value then the time required also increases linearly.

*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**.*

Totally it takes **'4n+4'** units of time to complete its execution and it is ***Linear Time Complexity***.

# Time Space Trade Off

In computer science, a space-time or time-memory tradeoff is a way of solving a problem in:

- Less time by using more memory) or,
- By solving a problem in very little space by spending a long time.

## Types of Trade Off

1) Compressed / Uncompressed Data
2) Re-Rendering / Stored Images
3) Smaller Code / Loop Unrolling
4) Lookup Table / Recalculation

### Compressed / Uncompressed Data

A space -time trade off can be applied to the problem of data storage, if data is stored uncompressed, it takes more space but less time. And if the data is stored compressed, it takes less space but more time to run the decompression algorithm.

### Re-Rendering / Stored Images

Storing only the source and rendering it as an image every time the page is requested would be trading time for space. **It will take more time but less space.** Storing the images would be trading space for time. **It will take more space but less time.**

### Smaller Code (with loop) / Larger Code (without loop)

Smaller code occupies **less space** in memory but it requires **high computation** time which is required for jumping back to the beginning of the loop at the end of each iteration.

Larger code or loop unrolling can be traded for higher program speed. It occupies **more space** in memory but requires **less computation time**.

### Lookup Table / Recalculation

In lookup table, an implementation can include the entire table which reduces computing time but increases the amount of memory needed. It can recalculate i.e. compute table entries as needed, increasing computing time but reducing memory requirements.

**Example: More time, Less space**

```
int a,b;
printf("enter value of a \n");
```

```
scanf("%d",&a);
printf("enter value of b \n");
scanf("%d",&b);
b=a+b;
printf("output is:%d",b);
```

**Example: More Space, Less time**

```
int a,b,c;
printf("enter value of a,b and c");
scanf("%d%d%d",&a,&b,&c);
printf("output is:%d",c=a+b);
```

# Pointer

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

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

A pointer variable takes 2 Bytes in memory, because the pointer variable holds the address of a variable and the address of memory location always in unsigned integer. An unsigned integer takes 2 Bytes; therefore, pointer occupies 2 Bytes.

**Declaration:**

```
data_type *variable_name;
```

**For example:**

```
int *p, *q;
```

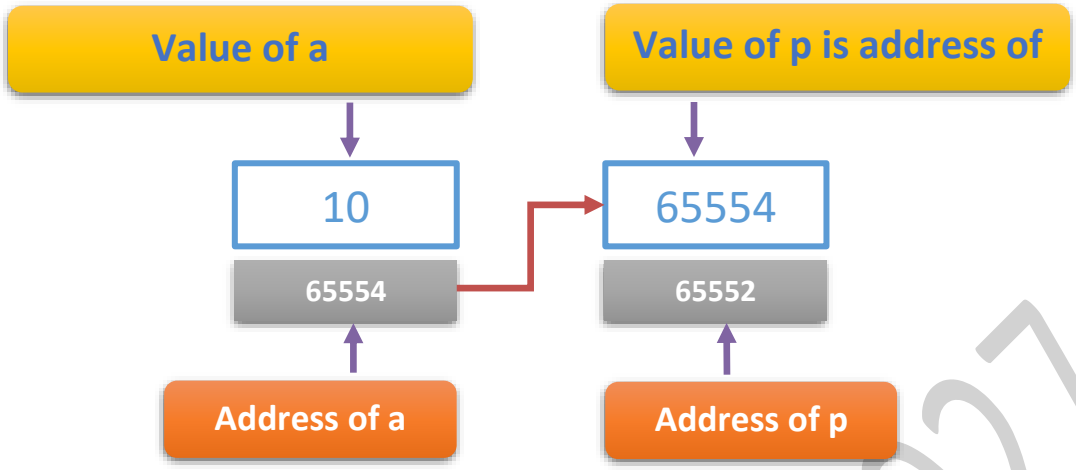
While learning about pointer we have to concentrate on two main things about pointer. These are –

**& - address of <variablename>**

*** - value of <variablename>**

**For example:** Consider there we have an integer variable and another is pointer variable. Let see following explanation –

```
int a=10;
int *p;
p=&a;
```




**a** = 10

**&a** = 65554

**p** = 65554

**&p** = 65552

***p** = 10

**For Example:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a;
    int *p;
    clrscr();
    printf("Enter a number for variable a: ");
    scanf("%d",&a);
    p=&a;
    printf("The value of a is: %d",a);
    printf("\nThe address of a is: %u",&a);
    printf("\n The value of p(holds the address of a) is: %u",p);
    printf("\n The address of p is: %u",&p);
    printf("\n The value of *p(value on address that holds by p) is: %d",*p);
    getch();
}
```

**OUTPUT:**

```
Enter a number for variable a: 25
The value of a is: 25
The address of a is: 65524
The value of p (holds the address of a) is: 65524
The address of p is: 65522
The value of *p(value on address that holds by p) is: 25
```

# Pointer Details

## Pointer arithmetic

There are four arithmetic operators that can be used in pointers: ++, --, +, -

## Array of pointers

We can define arrays to hold a number of pointers.

## Pointer to pointer

C allows us to have pointer on a pointer and so on.

## Passing pointers to functions in C

Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.

## Return pointer from functions in C

C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.

# Structure

A structure is a composite data type (User Defined Data Type) that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure. In other words, we can say that Structure is a group of different data items.

In 'C' programming arrays allow us to define variables that can hold several data items of the same kind but **structure** is another **user defined data type** available in C programming, which allows us to combine data items of different kinds.

As we mention above Structures are used to represent a record. Suppose we want to keep track of our books in a library. For example –

- Title
- Author
- Subject
- Book ID

So, store these kinds of information we use structure.

# Advantages

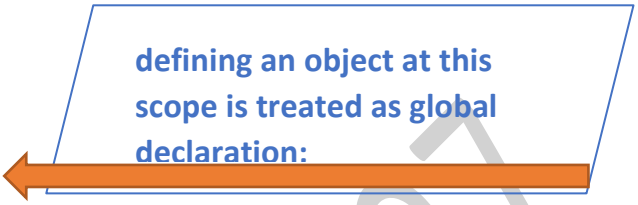
- It can hold variables of different data types.
- We can create objects containing different types of attributes.
- It allows us to re-use the data layout across programs.
- It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

## Defining a Structure

To define a structure, we use the **struct** statement. The **struct** statement defines a new data type, with more than one member for your program. Each member definition is a normal variable definition, such as `int` i; or `float` f; or any other valid variable definition. The syntax of the **struct** statement is following –

**Syntax:**

```
struct <structure tag name>
{
      member definition(s);
} <one or more structure Objects>;
```




## Defining a Structure Variable/Object

At the end of the structure's definition, **before the final semicolon, we can specify one or more structure variables but it is optional.** We can declare structure variable according to following methods –

1) **As Global Scope**
2) **As Local Scope**

### 1) As Global Scope

When we are defining variable of a structure as global scope then it is accessible to all functions ( main() and other user-defined function) equally.

**Syntax:**

| **Outside of all functions** | **Before the final semicolon of structure –** |
|---|---|
| `struct <tag name>`<br>`{`<br>`  members;`<br>`};`<br>`struct <tag name> <variable name>;` | `struct <tag name>`<br>`{`<br>`  members;`<br>`}struct <tag name> <variable name>;` |

### 2) As Local Scope

If we define a structure variable as local scope or in a function body then it will not be accessible outside of that function.

**Syntax:**

```
struct <tag name>
{
      members;
}
void main()
{
      struct <tag name> <variable name>;
}
```

## Accessing Structure Data Member

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure:

**For Example:**

**Que 118: WAP to define structure for book information and input the information about book then display it.**

```
#include<stdio.h>
#include<conio.h>
struct Books
{
    char t[50];
    char au[50];
    char s[100];
    int id;
};
void main()
{
    struct Books B;
    clrscr();
    printf("Enter book Title: ");
    gets(B.t);
    fflush(stdin);
    printf("Enter Author: ");
    scanf("%s",B.au);
    printf("Enter Subject: ");
    scanf("%s",B.s);
    printf("Enter Book ID: ");
    scanf("%d",&B.id);
    printf("Book Details are -");
    printf("Book Title: %s\n", B.t);
    printf("Book Author: %s\n", B.au);
    printf("Book Subject: %s\n", B.s);
    printf("Book ID: %d\n", B.id);
    getch();
}
```

**OUTPUT:**

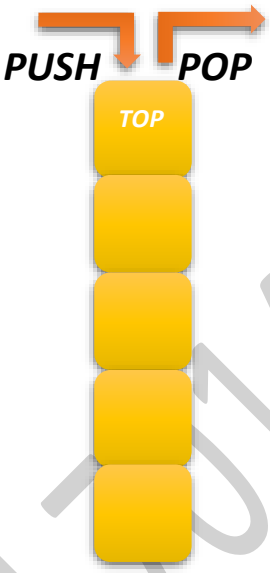
```
Enter Book Title: C Language
Enter Author: Unknown
Enter Subject: C
Enter Book ID: 231
Book Details are following -
Book Title: C Language
Book Author: Unknown
Book Subject: C
Book ID: 231
```

# Stack

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "top". That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on ***LIFO (Last In First Out)*** principle.

## Stack Representation

In a stack, the insertion operation is performed using a function called "***push***" and deletion operation is performed using a function called "***pop***".




In the figure, **PUSH** and **POP** operations are performed at a top position in the stack. *That means, both the insertion and deletion operations are performed at one end (at Top).*

A stack data structure can be defined as follows:

---

*Stack is a linear data structure in which the operations are performed based on **LIFO** principle.*

*A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle.*

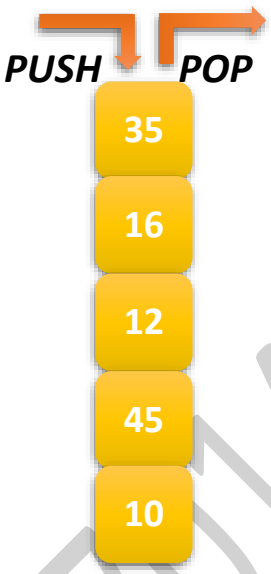
---

**Example:** If we want to create a stack by inserting 10, 45, 12, 16 and 35. Then 10 becomes the bottom-most element and 35 is the topmost element. The last inserted element 35 is at Top of the stack as shown in the image below.

## Primitive Operations on a Stack

The following operations are performed on the stack...

1) **Push (To insert an element on to the stack)**
2) **Pop (To delete an element from the stack)**




3) **Display (To display elements of the stack)**

## Implementation of Stack

Stack data structure can be implemented in two ways. They are as follows:

1) Using Array
2) Using Linked List

When a stack is implemented using an array, that stack can organize an only limited number of elements. When a stack is implemented using a linked list, that stack can organize an unlimited number of elements.

### 1) Stack Using Array

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one-dimensional array of specific size and insert or delete the values into that array by using ***LIFO*** principle with the help of a variable called ***'top'***. Initially, the top is set to ***-1***. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

A stack can be implemented using array as follows:

Before implementing actual operations, first follow the below steps to create an empty stack.

**Step 1** - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

**Step 2** - Declare all the **functions** used in stack implementation.

**Step 3** - Create a one-dimensional array with fixed size **(int stack[SIZE])**

**Step 4** - Define an integer variable **'top'** and initialize with **'-1'**. **(int top = -1)**

**Step 5** - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

## push(value) - Inserting value into the stack

In a stack, **push()** is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack:

**Step 1** - Check whether **stack** is **FULL. (top == SIZE-1)**

**Step 2** - If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.

**Step 3** - If it is **NOT FULL**, then increment **top** value by one **(top++)** and set **stack[top]** to value **(stack[top] = value**).

## pop() - Delete a value from the Stack

In a stack, **pop()** is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack:

**Step 1** - Check whether **stack** is **EMPTY**. **(top == -1)**

**Step 2** - If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

**Step 3** - If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one **(top--)**.

## display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack:

**Step 1** - Check whether stack is **EMPTY. (top == -1)**

**Step 2** - If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.

**Step 3** - If it is **NOT EMPTY**, then define a variable **'i'** and initialize with **top**. Display **stack[i]** value and decrement **i** value by one **(i--)**.

**Step 4** - Repeat above step until **i** value becomes **'0'**.

**C Program: Implementation of Stack using Array**

```
#include <stdio.h>
#include <conio.h>
#define SIZE 10
void push(int);
void pop();
void display();
int stack[SIZE], top = -1;
void main()
{
    int value, choice;
    while (1)
    {
        printf("\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);
                  break;
              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!!!\nPress Enter to continue\n");
        getch();
    }
    else
    {
        printf("\nDeleted : %d", stack[top]);
        top--;
    }
}
void display()
{
    if (top == -1)
    {
        printf("\nStack is Empty!!!\nPress Enter to continue\n");
        getch();
    }
    else
    {
        int i;
        printf("\nStack elements are:\n");
        for (i = top; i >= 0; i--)
            printf("%d\n", stack[i]);
    }
}
```

**OUTPUT:**

```
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 23
Insertion Success!!!
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion Success!!!
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements are:
23      12
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
```

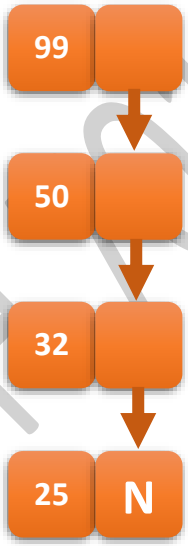
## 2) Stack Using Linked List

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use.

A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as **'top'** element. That means every newly inserted element is pointed by **'top'**. Whenever we want to remove an element from the stack, simply remove the node which is pointed by **'top'** by moving **'top'** to its previous node in the list. The **next** field of the first element must be always **NULL**.

**Example:**




In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

**Step 1** - Include all the header files which are used in the program. And declare all the user defined functions.

**Step 2** - Define a 'Node' structure with two members data and next.

**Step 3** - Define a Node pointer 'top' and set it to NULL.

**Step 4** - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

## push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

**Step 1** - Create a **newNode** with given value.

**Step 2** - Check whether stack is **Empty (top == NULL)**

**Step 3** - If it is **Empty**, then set **newNode → next = NULL.**

**Step 4** - If it is **Not Empty**, then set **newNode → next = top**.

**Step 5** - Finally, set **top** = **newNode**.

## pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack:

**Step 1** - Check whether **stack** is **Empty (top == NULL)**.

**Step 2** - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

**Step 3** - If it is **Not Empty**, then define a Node pointer ***'temp'*** and set it to ***'top'***.

**Step 4** - Then set **'top = top → next'.**

**Step 5** - Finally, delete **'temp'. (free(temp))**.

## display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack:

**Step 1** - Check whether **stack** is **Empty (top == NULL)**.

**Step 2** - If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.

**Step 3** - If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.

**Step 4** - Display **'temp → data --->'** and move it to the **next** node. Repeat the same until temp reaches to the first node in the **stack**. **(temp → next != NULL).**

**Step 5** - Finally! Display **'temp → data ---> NULL'**.

**C Program: Implementation of Stack using Linked List**

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *top = NULL;
void push(int);
void pop();
void display();
void main()
{
    int choice, value;
    printf("\n:: Stack using Linked List ::\n");
    while (1)
    {
        printf("\n****** Menu ******\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("Enter 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!!! Please try again!!!\n"
     );
        }
    }
}
void push(int value)
{
```

```
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = value;
    if (top == NULL)
    {
        newNode->next = NULL;
    }
    else
    {
        newNode->next = top;
    }
    top = newNode;
    printf("\nInsertion is Success!!!\n");
}
void pop()
{
    if (top == NULL)
    {
        printf("\nStack is Empty!!!\nPress Enter to continue\n");
        getch();
    }
    else
    {
        struct node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
        free(temp);
    }
}
void display()
{
    if (top == NULL)
    {
        printf("\nStack is Empty!!!\n Press Enter to continue\n");
        getch();
    }
    else
    {
        struct node *temp = top;
        while (temp->next != NULL)
        {
            printf("%d--->", temp->data);
            temp = temp->next;
        }
        printf("%d--->NULL", temp->data);
    }
}
```

**OUTPUT:**

```
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 23
Insertion Success!!!
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion Success!!!
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements are:
23--->12--->NULL
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
```

# Applications of Stack

There are following some important application of stacks are given following:

## Expression Conversion

An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

## Infix to Postfix or Infix to Prefix Conversion –

The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions. These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.

## Expression Evaluation

Stack is used to evaluate prefix, postfix and infix expressions.

## Postfix or Prefix Evaluation –

After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.

## Syntax Parsing

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

## Backtracking

Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

## Parenthesis Checking

Stack is used to check the proper opening and closing of parenthesis.

## String Reversal

Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

## Function Call

Stack is used to keep information about the active functions or subroutines.

# Expressions

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows:

*An expression is a collection of operators and operands that represents a specific value.*

In above definition, ***operator*** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

***Operands*** are the values on which the ***operators*** can perform the task. Here operand can be a direct value or variable or address of memory location.

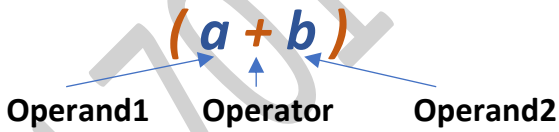
## Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows:

1) Infix Expression
2) Postfix Expression
3) Prefix Expression

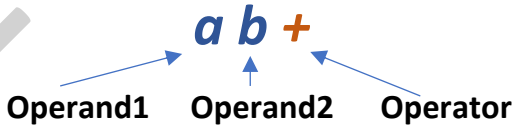
### 1) Infix Expression

In infix expression, operator is used in between the operands. The general structure of an Infix expression is as follows:



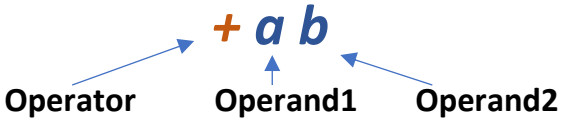

### 2) Postfix Expression

In postfix expression, operator is used after operands. We can say that **"Operator follows the Operands"**. The general structure of Postfix expression is as follows:




### 3) Prefix Expression

In prefix expression, operator is used before operands. We can say that **"Operands follows the Operator"**. The general structure of Prefix expression is as follows:



Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix, Infix to Prefix, Prefix to Postfix** and vice versa.

## Priority and Associativity of Operators

Before performing any expression conversion, we should know about the priority and associativity of operators. Please have a look at following table to understand it:

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | `++ --` | Suffix/postfix increment and decrement | Left-to-right |
| | `()` | Function call | |
| | `[]` | Array subscripting | |
| | `.` | Structure and union member access | |
| | `->` | Structure and union member access through pointer | |
| 2 | `++ --` | Prefix increment and decrement | Right-to-left |
| | `+ -` | Unary plus and minus | |
| | `! ~` | Logical NOT and bitwise NOT | |
| | `(type)` | Type cast | |
| | `*` | Indirection (dereference) | |
| | `&` | Address-of | |
| | `sizeof` | Size-of | |
| 3 | `* / %` | **Multiplication, division, and remainder** | Left-to-right |
| 4 | `+ -` | **Addition and subtraction** | |
| 5 | `<< >>` | Bitwise left shift and right shift | |
| 6 | `< <=` | For relational operators < and ≤ respectively | |
| | `> >=` | For relational operators > and ≥ respectively | |
| 7 | `== !=` | For relational = and ≠ respectively | |
| 8 | `&` | Bitwise AND | |
| 9 | `^` | Bitwise XOR (exclusive or) | |
| 10 | `\|` | Bitwise OR (inclusive or) | |
| 11 | `&&` | Logical AND | |
| 12 | `\|\|` | Logical OR | |
| 13 | `?:` | Ternary conditional | Right-to-Left |
| 14 | `=` | Simple assignment | |
| | `+= -=` | Assignment by sum and difference | |
| | `*= /= %=` | Assignment by product, quotient, and remainder | |
| | `<<= >>=` | Assignment by bitwise left shift and right shift | |
| | `&= ^= \|=` | Assignment by bitwise AND, XOR, and OR | |
| 15 | `,` | Comma | Left-to-right |

***Table: Operator precedence & Associativity***

## Infix to Postfix Conversion

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure:

- Find all the operators in the given Infix Expression.
- Find the order of operators evaluated according to their Operator precedence.
- Convert each operator into required type of expression (Postfix or Prefix) in the same order.

To convert Infix Expression into Postfix Expression using a stack data structure, we can use the following rules/steps:

1) Read all the symbols one by one from **left to right** in the given Infix Expression.
2) **If the reading symbol is operand, then directly print it to the result (Output).**
3) **If the reading symbol is operator, then Push it on to the top of stack.**
4) **If reading symbol is operator and If it has same priority in context of previous operator then we POP out the previous operator first then we will PUSH last scanned operator. For example: if - comes after + then + will POP out then – will PUSHED.**
5) **If reading symbol is operator and If it has higher priority than previous operator that has lower priority then it can stay with it. For example: if * comes after + then it can stay with +.**
6) **If reading symbol is operator and If it has lower priority than previous operator that has higher priority, then we will POP the existing higher priority operator from stack. For example: if - comes after / then / will POPPED from stack than – will Pushed.**
7) If the reading symbol is left parenthesis '(', then Push it on to the Stack.
8) **If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.**
9) **If the reading symbol is operator (+ , - , * , / etc.), then Push it on to the Stack.** However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

**Example: Let's consider following expression for conversion of INFIX to POSTFIX**

**Infix Expression: A+B*C-D/E+F-G**

| Step | Character Scanned | Stack | Postfix |
|---|---|---|---|
| 1 | A | Empty | A |
| 2 | + | + | A |
| 3 | B | + | A B |
| 4 | * | + * | A B |
| 5 | C | + * | A B C |
| 6 | - | - | A B C * + |
| 7 | D | - | A B C * + D |
| 8 | / | - / | A B C * + D |

| | | | |
|---|---|---|---|
| 9 | E | - / | A B C * + D E |
| 10 | + | + | A B C * + D E / - |
| 11 | F | + | A B C * + D E / - |
| 12 | - | - | A B C * + D E / - + |
| 13 | G | - | A B C * + D E / - + G |
| POSTFIX EXPRESSION | | | A B C * + D E / - + G - |

*Table: Infix to Postfix Conversion*

## Postfix to Infix Conversion

Now let us reverse the above conversion to verify that we are on right track:

| Step | Character Scanned (Postfix) | Infix |
|---|---|---|
| 1 | A | A |
| 2 | B | A B |
| 3 | C | A B C |
| 4 | * | A B * C |
| 5 | + | A + B * C |
| 6 | D | A + B * C D |
| 7 | E | A + B * C D E |
| 8 | / | A + B * C D / E |
| 9 | - | A + B * C - D / E |
| 10 | F | A + B * C - D / E F |
| 11 | + | A + B * C - D / E + F |
| 12 | G | A + B * C - D / E + F G |
| 13 | - | A + B * C - D / E + F – G |

*Table: Postfix to Infix Conversion*

## Infix to Prefix Conversion

As we convert infix to postfix, we can also convert infix to prefix also. To do so we have to follow some rules/steps, that are given following:

1) First reverse the given infix expression.
2) Scan all the Symbols and operands one by one from **left to the right.**
3) **If the reading symbol is operand, then directly print it to the result (Output).**
4) **If the reading symbol is operator, then Push it on to the top of stack.**
5) **If reading symbol is operator and If it has same priority in context of previous operator then it can stay with existing operator. For example: if - comes after + then + will be PUSHED at the top of stack.**
6) **If reading symbol is operator and If it has higher priority than previous operator then all the lower priority operator will POP out one by one then higher priority operator will PUSED at top of the stack. For example: if * comes after + then + will POP out from stack and then * will PUSHED.**

7) **If reading symbol is operator and If it has lower priority than previous operator then it will PUSHED at the top of the stack. For example: if + comes after / then + will be PUSHED at the top of the stack.**
8) If the reading symbol is left parenthesis '(', then Push it on to the Stack.
9) **If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.**
10) **If the reading symbol is operator (+ , - , * , / etc.), then Push it on to the Stack.** However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

**Example: Let's consider following expression for conversion of INFIX to PREFIX**

**Infix Expression: A – B * C + D / E + F - G**

Reversing the Infix Expression

**G – F + E / D + C * B - A**

| Step | Character Scanned | Stack | Postfix |
|---|---|---|---|
| 1 | G | Empty | G |
| 2 | - | - | G |
| 3 | F | - | F G |
| 4 | + | - + | E F G |
| 5 | E | - + | E F G |
| 6 | / | / | - + E F G |
| 7 | D | / | D - + E F G |
| 8 | + | / + | D - + E F G |
| 9 | C | / + | C D - + E F G |
| 10 | * | / * | + C D - + E F G |
| 11 | B | / * | B + C D - + E F G |
| 12 | - | / * - | B + C D - + E F G |
| 13 | A | / * - | A B + C D - + E F G |
| INFIX TO PREFIX | | | / * - A B + C D - + E F G |

*Table: Infix to Postfix Conversion*

## Prefix to Infix Conversion

Now let us reverse the above conversion to verify that we are on right track:

| Step | Character Scanned (Prefix) | Infix (Reversed) |
|---|---|---|
| 1 | G | G |
| 2 | F | G F |
| 3 | E | G F E |
| 4 | + | G F + E |
| 5 | - | G - F + E |

| | | |
|---|---|---|
| 6 | D | G - F + E D |
| 7 | C | G - F + E D C |
| 8 | + | G - F + E D + C |
| 9 | B | G - F + E D + C B |
| 10 | A | G - F + E D + C B A |
| 11 | - | G - F + E D + C B - A |
| 12 | * | G - F + E D + C * B - A |
| 13 | / | G - F + E / D + C * B - A |

*Table: Postfix to Infix Conversion*

Now reverse the expression to get infix Expression:

G – F + E / D + C * B – A

A – B * C + D / E + F - G

Reversing Back to get the Infix Expression

## Recursion

In C programming language, function calls can be made from the main() function, other functions or from the same function itself. The recursive function is defined as follows:

> *A function called by itself is called **recursive** function.*

***The recursive functions should be used very carefully because, when a function called by itself it enters into the infinite loop. And when a function enters into the infinite loop, the function execution never gets completed. We should define the condition to exit from the function call so that the recursive function gets terminated.***

When a function is called by itself, the first call remains under execution till the last call gets invoked. Every time when a function call is invoked, the function returns the execution control to the previous function call.

**Example:**

```
#include <stdio.h>
#include <conio.h>
int factorial(int);
void main()
{
    int fact, n;
    printf("Enter any positive integer: ");
    scanf("%d", &n);
    fact = factorial(n);
    printf("\nFactorial of %d is %d\n", n, fact);
}
```

```
int factorial(int n)
{
    int t = 1;
    if (t < n)
    {
        t = n * factorial(n - 1); // recursive function call
    }
    return t;
}
```

```
Enter any positive integer: 5
Factorial of 5 is 120
```

Output:

In the above example program, the **factorial**() function call is initiated from main() function with the value 5. Inside the **factorial()** function, the function calls **factorial(5)**, **factorial(4)**, **factorial(3)**, **factorial(2)**, **factorial(1)** and **factorial(0)** are called recursively. In this program execution process, the function call **factorial(5)** remains under execution till the execution of function calls **factorial(4)**, **factorial(3)**, **factorial(2)**, **factorial(1)** and **factorial(0)** gets completed. Similarly the function call **factorial(4)** remains under execution till the execution of function calls **factorial(3)**, **factorial(2)**, **factorial(1)** and **factorial(0)** gets completed. In the same way the function call **factorial(3)** remains under execution till the execution of function call **factorial(2)**, **factorial(1)** and **factorial(0)** gets completed. And this process will remain calling function recursively until the final call is made and the condition get false.

## Queue

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as ***'rear'*** and the deletion operation is performed at a position which is known as ***'front'***. In queue data structure, the insertion and deletion operations are performed based on ***FIFO (First In First Out)*** principle.


In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".

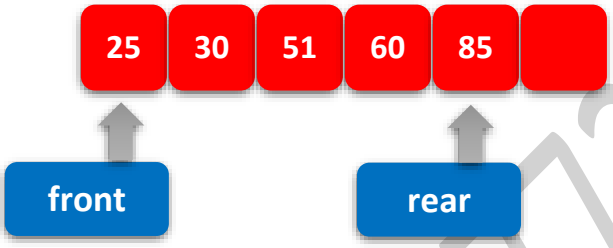
*Queue data structure is a linear data structure in which the operations are performed based on **FIFO** principle.*

Queue data structure can be defined as follows:

A queue data structure can also be defined as

**Example:** Queue after inserting 25, 30, 51, 60 and 85.

After inserting five elements:




*Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on **FIFO** principle.*

## Primitive Operations on a Queue

The following operations are performed on a queue data structure:

1) **enQueue(value) - (To insert an element into the queue)**
2) **deQueue() - (To delete an element from the queue)**
3) **display() - (To display the elements of the queue)**

## Implementation of Queue Data Structure

Queue data structure can be implemented in two ways. They are as follows:

1) Using Array
2) Using Linked List

When a queue is implemented using an array, that queue can organize an only limited number of elements. When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

### 1) Queue Using Array

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one-dimensional array of specific

size and insert or delete the values into that array by using ***FIFO (First In First Out)*** principle with the help of variables ***'front'*** and ***'rear'***.

Initially both ***'front'*** and ***'rear'*** are set to **-1**. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Before we implement actual operations, first follow the below steps to create an empty queue:

**Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

**Step 2 -** Declare all the **user defined functions** which are used in queue implementation.

**Step 3 -** Create a one dimensional array with above defined **SIZE (int queue[SIZE])**

**Step 4 -** Define two integer variables **'front'** and **'rear'** and initialize both with **'-1'**. **(int front = -1, rear = -1)**

**Step 5 -** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

## enQueue(value) - Inserting value into the queue

In a queue data structure, **enQueue()** is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The **enQueue()** function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue:

**Step 1 -** Check whether queue is **FULL**. **(rear == SIZE-1)**

**Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

**Step 3 -** If it is **NOT FULL**, then increment **rear** value by one **(rear++)** and set **queue[rear] = value**.

## deQueue() - Deleting a value from the Queue

In a queue data structure, **deQueue()** is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The **deQueue()** function does not take any value as parameter. We can use the following steps to delete an element from the queue:

**Step 1 -** Check whether queue is **EMPTY**. **(front == rear)**

**Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

**Step 3 -** If it is **NOT EMPTY**, then increment the **front** value by one **(front ++)**. Then display **queue[front]** as deleted element. Then check whether both front and rear are equal **(front == rear)**, if it **TRUE**, then set both front and rear to **'-1' (front = rear = -1)**.

## display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue:

**Step 1 -** Check whether queue is **EMPTY**. **(front == rear)**

**Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.

**Step 3 -** If it is **NOT EMPTY**, then define an integer variable **'i'** and set **'i = front+1'**.

**Step 4 -** Display **'queue[i]'** value and increment **'i'** value by one **(i++)**. Repeat the same until **'i'** value reaches to **rear (i <= rear).**

**C Program: Implementation of Queue using Array**

```
#include <stdio.h>
#include <conio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;
void main()
{
    int value, choice;
    while (1)
    {
        printf("\n\n***** MENU *****\n");
        printf("1. Insertion\n2. Deletion\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);
                enQueue(value);
                break;
            case 2:
                deQueue();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
                break;
```

```
                default:
                    printf("\nWrong selection!!! Try again!!!");
            }
        }
    }
    void enQueue(int value)
    {
        if (rear == SIZE - 1)
        {
            printf("\nQueue is Full!!! Insertion is not possible!!!");
        }
        else
        {
            if (front == -1)
            {
                front = 0;
            }
            rear++;
            queue[rear] = value;
            printf("\nInsertion success!!!");
        }
    }
    void deQueue()
    {
        if (front == rear)
        {
            printf("\nQueue is Empty!!! Deletion is not possible!!!");
        }
        else
        {
            printf("\nDeleted : %d", queue[front]);
            front++;
            if (front == rear)
            {
                front = rear = -1;
            }
        }
    }
    void display()
    {
        if (rear == -1)
        {
            printf("\nQueue is Empty!!!");
        }
        else
        {
            int i;
            printf("\nQueue elements are:\n");
            for (i = front; i <= rear; i++)
            {
```

```
            printf("%d\t", queue[i]);
        }
    }
}
```

**OUTPUT:**

```
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 23
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
Queue elements are:
23     12
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 4
```

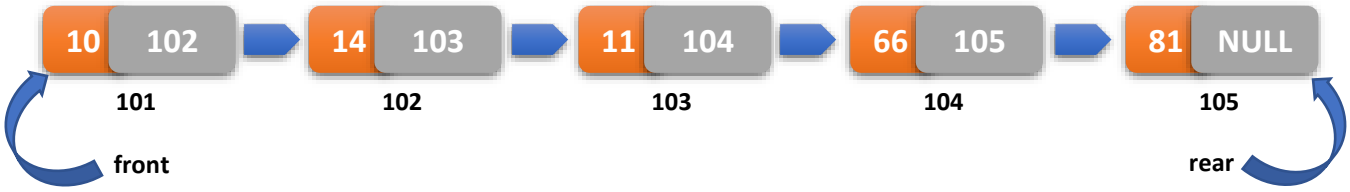
## 2) Queue Using Linked List

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use.

A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by **'rear'** and the first node is always pointed by **'front'**.

**Example**




In above example, the last inserted node is 50 and it is pointed by **'rear'** and the first inserted node is 10 and it is pointed by **'front'**. The order of elements inserted is 10, 15, 22 and 50.

To implement queue using linked list, we need to set the following things before implementing actual operations.

**Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.

**Step 2 -** Define a **'Node'** structure with two members **data** and **next**.

**Step 3 -** Define two Node pointers **'front'** and **'rear'** and set both to **NULL**.

**Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

## enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue:

**Step 1 -** Create a **newNode** with given value and set **'newNode → next'** to **NULL**.

**Step 2 -** Check whether queue is **Empty (rear == NULL)**

**Step 3 -** If it is **Empty** then, set **front = newNode** and **rear = newNode**.

**Step 4 -** If it is Not **Empty** then, set **rear → next = newNode** and **rear = newNode**.

## deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue:

**Step 1 -** Check whether queue is **Empty (front == NULL).**

**Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function

**Step 3 -** If it is Not **Empty** then, define a **Node** pointer **'temp'** and set it to **'front'**.

**Step 4 -** Then set **'front = front → next'** and delete **'temp' (free(temp))**.

## display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

**Step 1 -** Check whether queue is **Empty (front == NULL)**.

**Step 2 -** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.

**Step 3 -** If it is Not **Empty** then, define a Node pointer **'temp'** and initialize with **front**.

**Step 4 -** Display **'temp → data --->'** and move it to the **next** node. Repeat the same until **'temp'** reaches to **'rear' (temp → next != NULL)**.

**Step 5 -** Finally! Display **'temp → data ---> NULL'**.

**C Program: Implementation of Queue using Linked List**

```
#include <stdio.h>
#include <conio.h>
struct Node
{
    int data;
    struct Node *next;
} *front = NULL, *rear = NULL;

void insert(int);
void delete ();
void display();

void main()
{
    int choice, value;
    clrscr();
    while (1)
    {
        printf("\n****** MENU ******\n");
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("Enter the value to be insert: ");
            scanf("%d", &value);
            insert(value);
            break;
        case 2:
            delete ();
            break;
        case 3:
            display();
            break;
```

```
            case 4:
                exit(0);
                break;
            default:
                printf("\nWrong selection!!! Please try again!!!\n");
            }
        }
    }

    void insert(int value)
    {
        struct Node *newNode;
        newNode = (struct Node *)malloc(sizeof(struct Node));
        newNode->data = value;
        newNode->next = NULL;
        if (front == NULL)
        {
            front = rear = newNode;
        }

        else
        {
            rear->next = newNode;
            rear = newNode;
        }
        printf("\nInsertion is Success!!!\n");
    }
    void delete ()
    {
        if (front == NULL)
        {
            printf("\nQueue is Empty!!!\n");
        }
        else
        {
            struct Node *temp = front;
            front = front->next;
            printf("\nDeleted element: %d\n", temp->data);
            free(temp);
        }
    }
    void display()
    {
        if (front == NULL)
        {
            printf("\nQueue is Empty!!!\n");
        }
        else
        {
            struct Node *temp = front;
```

```
        while (temp->next != NULL)
        {
            printf("%d--->", temp->data);
            temp = temp->next;
        }
        printf("%d--->NULL\n", temp->data);
    }
}
```

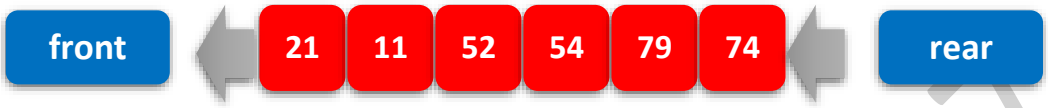
**OUTPUT:**

```
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 23
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 45
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
23--->12--->45--->NULL
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 4
```

## Circular Queue

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example, consider the queue below:

The queue after inserting all the elements into it is as follows:




Now consider the following situation after deleting three elements from the queue:


**Queue is full (even if three elements are deleted from front)**

This situation also says that Queue is Full and we cannot insert the new element because **'rear'** is still at last position. In the above situation, even though we have empty positions in the queue we cannot make use of them to insert the new element. This is the major problem in a normal queue data structure. To overcome this problem, we use a **circular queue** data structure.

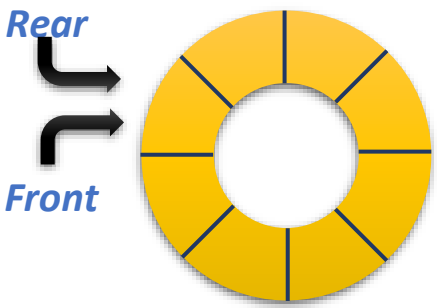
A Circular Queue can be defined as follows:

---

*A circular queue is a linear data structure in which the operations are performed based on **FIFO (First In First Out)** principle and the last position is connected back to the first position to make a circle.*

---

Graphical representation of a circular queue is as follows:



## Implementation of Circular Queue

To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.

**Step 1 -** Include all the header files which are used in the program and define a constant **'SIZE'** with specific value.

**Step 2 -** Declare all **user defined functions** used in circular queue implementation.

**Step 3 -** Create a one dimensional array with above defined **SIZE (int cQueue[SIZE])**

**Step 4 -** Define two integer variables **'front'** and **'rear'** and initialize both with **'-1'**. (**int front = -1, rear = -1)**

**Step 5 -** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

## enQueue(value) - Inserting value into the Circular Queue

In a circular queue, **enQueue()** is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The **enQueue()** function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue:

**Step 1 -** Check whether queue is **FULL**. **((rear == SIZE-1 && front == 0) || (front == rear+1))**

**Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

**Step 3 -** If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.

**Step 4 -** Increment **rear** value by one **(rear++)**, set **queue[rear] = value** and check **'front == -1'** if it is **TRUE**, then set **front = 0**.

## deQueue() - Deleting a value from the Circular Queue

In a circular queue, **deQueue()** is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The **deQueue()** function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular queue:

**Step 1 -** Check whether queue is **EMPTY**. **(front == -1 && rear == -1)**

**Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

**Step 3 -** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the front value by one **(front ++)**. Then check whether **front == SIZE**, if it is **TRUE**, then set **front =**

**0**. Then check whether both **front -1** and **rear** are equal **(front -1 == rear)**, if it **TRUE**, then set both **front** and **rear** to **'-1' (front = rear = -1)**.

## display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue:

**Step 1 -** Check whether queue is **EMPTY**. **(front == -1)**

**Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.

**Step 3 -** If it is **NOT EMPTY**, then define an integer variable **'i'** and set **'i = front'**.

**Step 4 -** Check whether **'front <= rear'**, if it is **TRUE**, then display **'queue[i]'** value and increment **'i'** value by one **(i++)**. Repeat the same until **'i <= rear'** becomes **FALSE**.

**Step 5 -** If **'front <= rear'** is **FALSE**, then display **'queue[i]'** value and increment **'i'** value by one **(i++)**. Repeat the same until **'i <= SIZE - 1'** becomes **FALSE**.

Step 6 - Set **i** to **0**.

Step 7 - Again display **'cQueue[i]'** value and increment **i** value by one **(i++)**. Repeat the same until **'i <= rear'** becomes **FALSE**.

**C Program: Implementation of Circular Queue**

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define SIZE 5
void enQueue(int);
void deQueue();
void display();
int cQueue[SIZE], front = -1, rear = -1;
void main()
{
    int choice, value;
    while (1)
    {
        printf("\n****** MENU ******\n");
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
              case 1:
                  printf("\nEnter the value to be insert:  ");
                  scanf("%d", &value);
                  enQueue(value);
                  break;
              case 2:
                  deQueue();
```

```
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("\nPlease select the correct choice!!!\n");
        }
    }
}
void enQueue(int value)
{
    if ((front == 0 && rear == SIZE - 1) || (front == rear + 1))
    {
        printf("\nCircular Queue is Full! Insertion not possible!!!\n");
    }
    else
    {
        if (rear == SIZE - 1 && front != 0)
        {
            rear = -1;
        }
        cQueue[++rear] = value;
        printf("\nInsertion Success!!!\n");
        if (front == -1)
        {
            front = 0;
        }
    }
}
void deQueue()
{
    if (front == -1 && rear == -1)
    {
        printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
    }
    else
    {
        printf("\nDeleted element : %d\n", cQueue[front++]);
        if (front == SIZE)
        {
            front = 0;
        }
        if (front - 1 == rear)
        {
            front = rear = -1;
        }
    }
}
```

```c
void display()
{
    if (front == -1)
    {
        printf("\nCircular Queue is Empty!!!\n");
    }
    else
    {
        int i = front;
        printf("\nCircular Queue Elements are : \n");
        if (front <= rear)
        {
            while (i <= rear)
            {
                printf("%d\t", cQueue[i++]);
            }
        }
        else
        {
            while (i <= SIZE - 1)
            {
                printf("%d\t", cQueue[i++]);
            }
            i = 0;
            while (i <= rear)
            {
                printf("%d\t", cQueue[i++]);
            }
        }
    }
}
```

**OUTPUT:**

```
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 23
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 44
Insertion Success!!!
```

```
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice:3
Circular Queue Elements are:
23      12      44
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice:2
Deleted element: 23
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice:2
Deleted element: 12
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice:3
Circular Queue Elements are:
44
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice:4
```

# Linked List

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "***Node***".

A ***Node*** contains two fields: **data** stored at that particular address and the **pointer** which contains the address of the next node in the memory. The last node of the list contains pointer to the **null**.

## Types of Linked List

Following are the various types of linked list.

1) Simple Linked List − Item navigation is forward only.
2) Doubly Linked List − Items can be navigated forward and backward.
3) Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

### 1) Linear/Single Linked List

Simply a list is a sequence of data, and the linked list is a sequence of data linked with

---

*Single linked list is a sequence of elements in which every element has link to its next element in the sequence.*

---

each other. The formal definition of a single linked list is as follows:

In any *single linked list*, the individual element is called as "**Node**". Every "**Node**" contains two fields, **data field**, and the **next field**. The **data field** is used to store actual value of the node and **next field** is used to store the address of next node in the sequence. The graphical representation of a node in a single linked list is as follows:

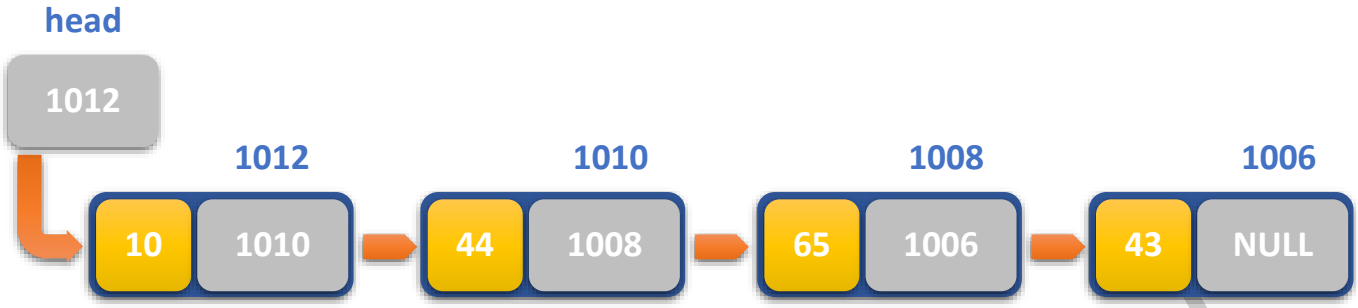
## Important Points to be Remember

- ***In a single linked list, the address of the first node is always stored in a reference node known as "front" or "head".***


- ***Always next part (reference part) of the last node must be NULL.***

## Representation of Linear Linked List

Linked list can be visualized as a chain of nodes, where every node point to the next node.




## Operations on Single Linked List

The following operations are performed on a Single Linked List

1) Insertion
2) Deletion
3) Display
4) Search
5) Sort

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

**Step 1 -** Include all the header files which are used in the program.

**Step 2 -** Declare all the user defined functions.

**Step 3 -** Define a Node structure with two members **data** and **next**

**Step 4 -** Define a Node pointer **'head'** and set it to **NULL**.

**Step 5 -** Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

## Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows:

1) Inserting at End of the list
2) Inserting at Beginning of the list
3) Inserting at Specific location in the list

### *1) Inserting at End of the list*

We can use the following steps to insert a new node at end of the single linked list:

Step 1 - Create a **temp** with given value and **temp → next** as **NULL**.

Step 2 - Check whether list is Empty **(head == NULL).**

Step 3 - If it is Empty then, set **head = temp**.

Step 4 - If it is Not Empty then, define a **node** pointer **ptr** and initialize with **head**.

Step 5 - Keep moving the **ptr** to its **next node** until it reaches to the last node in the list (until **ptr → next is equal to NULL**).

Step 6 - Set **ptr→ next = temp.**

**C Program: Inserting new nodes (Insert at last)**

```
#include <stdio.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head = NULL;
void insertlast(int n)
{
    struct node *ptr, *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = n;
    temp->next = NULL;
    if (head == NULL)
    {
        head = temp;
    }
    else
    {
        ptr = head;
        while (ptr->next != NULL)
        {
            ptr = ptr->next
        }
        ptr->next = temp;
    }
}
void display()
{
    struct node *ptr;
    if (head == NULL)
    {
        printf("There is nothing to display!!");
    }
    else
    {
        ptr = head;
        while (ptr != NULL)
```

```
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        }
    }
}
void main()
{
    int ch, n;
    while (1)
    {
        printf("\nEnter Your choice:\n1. Insert\n 2. Display \n0. Exit \nChoice:");
        scanf("%d", &ch);

        switch (ch)
        {
        case 0:
            exit(0);
            break;
        case 1:
            printf"Enter data to insert: ");
            scanf("%d", &n);
            insertlast(n);
            break;
        case 2:
            display();
            break;
        default:
            printf("Select a correct option and try again!!\n");
        }
    }
}
```

**OUTPUT:**

```
Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 1
Enter data to insert: 44
Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 1
Enter data to insert: 12
Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 1
Enter data to insert: 65
Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 2
44      12      65
```

## *2) Inserting at Beginning of the list*

We can use the following steps to insert a new node at beginning of the single linked list:

**Step 1 -** Create a **temp** with given value.

**Step 2 -** Set **temp→data = n.**

**Step 3-** Set **temp→next = head** and **head = temp.**

**C Program: Insert at beginning**

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head = NULL;
void insertbeg(int n)
{
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = n;
    temp->next = head;
    head = temp;
}
void disp()
{
    struct node *ptr;
    if (head == NULL)
    {
        printf("\nNothing to diplay!!");
    }
    else
    {
        ptr = head;
        printf("Linked List:\n");
        while (ptr != NULL)
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        }
    }
}
void main()
{
    int ch, n;
    while (1)
    {
```

```
        printf("\nEnter Your choice:\n1. Insert at Beginning \n2. Display \n0. E
xit\n");
        scanf("%d", &ch);
        switch (ch)
        {
              case 0:
                  exit(0);
                  break;
              case 1:
                  printf("Enter data to insert at beginning: ");
                  scanf("%d", &n);
                  insertbeg(n);
                  break;
              case 2:
                  disp();
                  break;
              default:
                  printf("\nSelect a correct option and try again :\n");
        }
    }
}
```

**OUTPUT:**

```
Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 1
Enter data to insert at beginning: 44

Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 1
Enter data to insert at beginning: 32

Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 1
Enter data to insert at beginning: 65

Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 2
65      32      44

Enter Your choice:
1. Insert
2. Display
0. Exit
Choice: 0
```

### *3) Inserting at Specific location in the list (After a Node)*

We can use the following steps to insert a new node after a node in the single linked list:

**Step 1 -** Create a **temp** with given value.

**Step 2 -** Check whether list is Empty **(head == NULL)**

**Step 3 -** If it is Empty then, set **newNode → next = NULL** and **head = newNode**.

**Step 4 -** If it is Not Empty then, define a node pointer **temp** and initialize with **head**.

**Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp → data** is equal to location, here location is the node value after which we want to insert the **newNode**).

**Step 6 -** Every time check whether temp is reached to last node or not. If it is reached to last node then display **"Given node is not found in the list!!! Insertion not possible!!!"** and terminate the function. Otherwise move the temp to next node.

**Step 7 -** Finally, Set **'newNode → next = temp → next'** and **'temp → next = newNode'**

**C Program: Insert at specific location**

```
#include <stdio.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
};

struct node *head = NULL;
void insertmid(int n, int l)
{
    int i;
    struct node *ptr, *temp;
    if (head == NULL)
    {
        printf("\nInsertion not possible in empty list");
    }
    else
    {
        temp = (struct node *)malloc(sizeof(struct node));
        temp->data = n;
        if (l == 1)
        {
            temp->next = head;
            head = temp;
        }
        else
```

```
        {
            i = 2;
            ptr = head;
            while (i < l && ptr->next != NULL)
            {
                i++;
                ptr = ptr->next;
            }
            temp->next = ptr->next;
            ptr->next = temp;
        }
    }
}
void disp()
{
    struct node *ptr;
    if (head == NULL)
    {
        printf("\nNothing to diplay!!");
    }
    else
    {
        ptr = head;
        printf("Linked List:\n");
        while (ptr != NULL)
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        }
    }
}
void insertlast(int n)
{
    struct node *ptr, *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = n;
    temp->next = NULL;
    if (head == NULL)
    {
        head = temp;
    }
    else
    {
        ptr = head;
        while (ptr->next != NULL)
        {
            ptr = ptr->next;
        }
        ptr->next = temp;
    }
```

```
}
void main()
{
    int ch, n, l;
    while (1)
    {
        printf("\nEnter Your choice:");
        printf("\n1. Insert\n2. Insert at specific location\n3. Display\n0. Exit\n");
        scanf("%d", &ch);
        switch (ch)
        {
              case 0:
                  exit(0);
              break;
              case 1:
                  printf("Enter data to insert: ");
                  scanf("%d", &n);
                  insertlast(n);
              break;
              case 2:
                  printf("Enter data to insert: ");
                  scanf("%d", &n);
                  printf("Enter location: ");
                  scanf("%d", &l);
                  insertmid(n,l);
              break;
              case 3:
                  disp();
              break;
              default:
                  printf("\nSelect a correct option and try again :\n");
        }
    }
}
```

**OUTPUT:**

```
Enter Your choice:
1. Insert
2. Insert at specific location
3. Display
0. Exit
Choice: 1
Enter data to insert: 44

Enter Your choice:
1. Insert
2. Insert at specific location
3. Display
0. Exit
Choice: 1
Enter data to insert: 21

Enter Your choice:
1. Insert
2. Insert at specific location
3. Display
0. Exit
Choice: 2
```

```
Enter data to insert: 26
Enter Location: 2

Enter Your choice:
1. Insert
2. Insert at specific location
3. Display
0. Exit
Choice: 3
Linked List:
44      26      21
```

## Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1) Deleting from Beginning of the list
2) Deleting from End of the list
3) Deleting a Specific Node

### *1) Deleting from Beginning of the list*

We can use the following steps to delete a node from beginning of the single linked list...

**Step 1 -** Check whether list is Empty **(head == NULL)**

**Step 2 -** If it is Empty then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is Not Empty then, define a Node pointer **'temp'** and initialize with **head**.

**Step 4 -** Check whether list is having only one node **(temp → next == NULL)**

**Step 5 -** If it is TRUE then set **head = NULL** and delete **temp** (Setting Empty list conditions)

**Step 6 -** If it is FALSE then set **head = temp → next**, and delete **temp**.

**C Program: Deleting from the beginning of the list**

```c
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head = NULL;
void insert(int n)
{
    struct node *ptr, *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = n;
```

```
    temp->next = NULL;
    if (head == NULL)
    {
        head = temp;
    }
    else
    {
        ptr = head;
        while (ptr->next != NULL)
        {
            ptr = ptr->next;
        }
        ptr->next = temp;
    }
}
void disp()
{
    struct node *ptr;
    if (head == NULL)
    {
        printf("\nNothing to diplay!!");
    }
    else
    {
        ptr = head;
        printf("Linked List:\n");
        while (ptr != NULL)
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        }
    }
}
void removeBeginning()
{
    if (head == NULL)
      printf("\nList is Empty!!!");
    else
    {
        struct node *temp = head;
        if (head->next == NULL)
        {
            head = NULL;
            free(temp);
            printf("\nNode deleted from Beginning!!!\n");
        }
        else
        {
            head = temp->next;
            free(temp);
```

```
            printf("\nNode deleted from Beginning!!!\n");
        }
    }
}
void main()
{
    int ch, n;
    while (1)
    {
        printf("\nEnter Your choice\n1. Insert\n2. Display\n3. Remove from the beginning \n0. Exit\n");
        scanf("%d", &ch);
        switch (ch)
        {
            case 0:
                exit(0);
                break;
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &n);
                insert(n);
                break;
            case 2:
                disp();
                break;
            case 3:
                removeBeginning();
                break;

            default:
                printf("\nSelect a correct option and try again :\n");
        }
    }
}
```

**OUTPUT:**

```
Enter Your choice:
1. Insert
2. Display
3. Remove from the beginning
0. Exit
1
Enter data to insert: 12
Enter Your choice:
1. Insert
2. Display
3. Remove from the beginning
0. Exit
1
Enter data to insert: 34
Enter Your choice:
1. Insert
2. Display
3. Remove from the beginning
0. Exit
```

```
3
Node deleted from Beginning!!!
Enter Your choice:
1. Insert
2. Display
3. Remove from the beginning
0. Exit
2
Linked List:
34
Enter Your choice:
1. Insert
2. Display
3. Remove from the beginning
0. Exit
0
```

## *2) Deleting from End of the list*

We can use the following steps to delete a node from end of the single linked list...

**Step 1 -** Check whether list is Empty **(head == NULL)**

**Step 2 -** If it is Empty then, display **'List is Empty!!! Deletion is not possible**' and terminate the function.

**Step 3 -** If it is Not Empty then, define two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**.

**Step 4 -** Check whether list has only one Node **(temp1 → next == NULL)**

**Step 5 -** If it is TRUE. Then, set **head = NULL** and delete **temp1**. And terminate the function. (Setting Empty list condition)

**Step 6 -** If it is FALSE. Then, set **'temp2 = temp1'** and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next == NULL**)

**Step 7 -** Finally, Set **temp2 → next = NULL** and delete **temp1**.

**C Program: Deleting from the end of the Linked List**

```
#include <stdio.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head = NULL;
void insert(int n)
{
```

```
    struct node *ptr, *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = n;
    temp->next = NULL;
    if (head == NULL)
    {
        head = temp;
    }
    else
    {
        ptr = head;
        while (ptr->next != NULL)
        {
            ptr = ptr->next;
        }
        ptr->next = temp;
    }
}
void disp()
{
    struct node *ptr;
    if (head == NULL)
    {
        printf("\nNothing to diplay!!");
    }
    else
    {
        ptr = head;
        printf("Linked List:\n");
        while (ptr != NULL)
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        }
    }
}
void removeEnd()
{
    if (head == NULL)
    {
        printf("\nList is Empty!!!\n");
    }
    else
    {
        ptr = head;

        if (ptr->next == NULL)
        {
            f = ptr;
            head = NULL;
            free(f);
```

```
        }
        else
        {
            ptr = head;
            while (ptr->next->next != NULL)
            {
                ptr = ptr->next;
            }
            f = ptr->next;
            printf("\nNode deleted from the last position\n");
            ptr->next = NULL;
            free(f);
            disp();
        }
    }
}
void main()
{
    int ch, n;
    while (1)
    {
        printf("\nEnter Your choice:\n1. Insert\n2. Display\n3. Remove Last Node
 \n0. Exit\n");
        scanf("%d", &ch);
        switch (ch)
        {
        case 0:
            exit(0);
            break;
        case 1:
            printf("Enter data to insert: ");
            scanf("%d", &n);
            insert(n);
            break;
        case 2:
            disp();
            break;
        case 3:
            removeEnd();
            break;
        default:
            printf("\nSelect a correct option and try again :\n");
        }
    }
}
```

**OUTPUT:**

```
Enter Your choice:
1. Insert
2. Display
3. Remove Last Node
0. Exit
1
Enter data to insert: 65

Enter Your choice:
1. Insert
2. Display
3. Remove Last Node
0. Exit
1
Enter data to insert: 43

Enter Your choice:
1. Insert
2. Display
3. Remove Last Node
0. Exit
1
Enter data to insert: 65

Enter Your choice:
1. Insert
2. Display
3. Remove Last Node
0. Exit
3

Node deleted from the last position
Linked List:
65      43
Enter Your choice:
1. Insert
2. Display
3. Remove Last Node
0. Exit
```

## *3) Deleting a Specific Node from the list*

We can use the following steps to delete a specific node from the single linked list...

**Step 1 -** Check whether list is Empty **(head == NULL)**

**Step 2 -** If it is Empty then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is Not Empty then, define two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**.

**Step 4 -** Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set **'temp2 = temp1'** before moving the **'temp1'** to its next node.

**Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

**Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

**Step 7 -** If list has only one node and that is the node to be deleted, then set **head = NULL** and delete temp1 **(free(temp1)).**

**Step 8 -** If list contains multiple nodes, then check whether temp1 is the first node in the list **(temp1 == head).**

**Step 9 -** If **temp1** is the first node then move the head to the next node **(head = head → next)** and delete **temp1**.

**Step 10 -** If temp1 is not first node then check whether it is last node in the **list (temp1 → next == NULL).**

**Step 11 -** If temp1 is last node then set **temp2 → next = NULL** and delete **temp1 (free(temp1))**.

**Step 12 -** If temp1 is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1 (free(temp1)).**

**C Program: Deleting a node from specific location**

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head = NULL;
void insert(int n)
{
```

```
    struct node *ptr, *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = n;
    temp->next = NULL;
    if (head == NULL)
    {
        head = temp;
    }
    else
    {
        ptr = head;
        while (ptr->next != NULL)
        {
            ptr = ptr->next;
        }
        ptr->next = temp;
    }
}
void disp()
{
    struct node *ptr;
    if (head == NULL)
    {
        printf("\nNothing to diplay!!");
    }
    else
    {
        ptr = head;
        printf("Linked List:\n");
        while (ptr != NULL)
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        }
    }
}
void dltSpc(int l)
{
    struct node *ptr, *f;
    int i, flg = 0;
    if (head == NULL)
    {
        printf("Nothing to delete!!!");
    }
    else
    {
        if (l == 1)
        {
            f = head;
            head = f->next;
```

```
            free(f);
        }
        else if (l <= countNode())
        {
            i = 2;
            ptr = head;
            while (i < l && ptr->next != NULL)
            {
                i++;
                ptr = ptr->next;
            }

            f = ptr->next;
            ptr->next = ptr->next->next;
            free(f);
            printf("One Node deleted!!!!\n");
        }
        else
        {
            printf("Desired location not found!! Try again!!\n");
        }
    }
    printf("\n");
    disp();
}
void main()
{
    int ch, n, l;
    while (1)
    {
        printf("\nEnter Your choice:\n1. Insert\n2. Display\n3. Delete from Spec
ific location \n0. Exit\n");
        scanf("%d", &ch);
        switch (ch)
        {
        case 0:
            exit(0);
            break;
        case 1:
            printf("Enter data to insert: ");
            scanf("%d", &n);
            insert(n);
            break;
        case 2:
            disp();
            break;
        case 3:
            printf("Enter location to delete data: ");
            scanf("%d", &l);
            dltSpc(l);
```

```
            break;
        default:
            printf("\nSelect a correct option and try again :\n");
        }
    }
}
```

**Output:**

```
Enter Your choice:
1. Insert
2. Display
3. Delete from Specific location
0. Exit
1
Enter data to insert: 21

Enter Your choice:
1. Insert
2. Display
3. Delete from Specific location
0. Exit
1
Enter data to insert: 34

Enter Your choice:
1. Insert
2. Display
3. Delete from Specific location
0. Exit
1
Enter data to insert: 56

Enter Your choice:
1. Insert
2. Display
3. Delete from Specific location
0. Exit
1
Enter data to insert: 45

Enter Your choice:
1. Insert
2. Display
3. Delete from Specific location
0. Exit
3
Enter location to delete data: 3
One Node deleted!!!!

Linked List:
21      34      45
Enter Your choice:
1. Insert
2. Display
3. Delete from Specific location
0. Exit
0
```

## Display

We can use the following steps to display the elements of a single linked list...

**Step 1 -** Check whether list is Empty **(head == NULL)**

**Step 2 -** If it is Empty then, display **'List is Empty!!!'** and terminate the function.

**Step 3 -** If it is Not Empty then, define a Node pointer **'temp'** and initialize with **head**.

**Step 4 -** Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

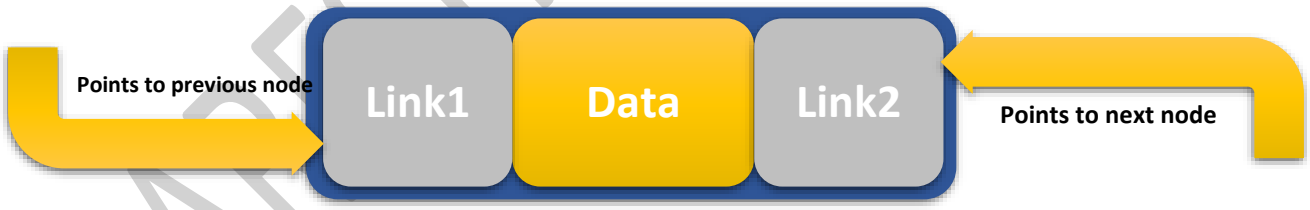
**Step 5 -** Finally display temp → data with arrow pointing to **NULL (temp → data ---> NULL)**.

## 2) Double Linked List

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we cannot traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

*Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.*

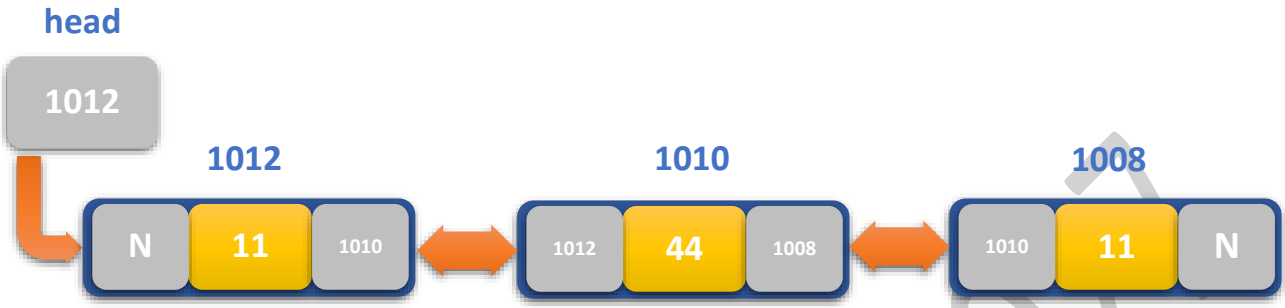
In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...




Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

## Representation of Doubly Linked List

We can represent doubly linked list as given following in which each node has two address part. Left-side address part points address of previous node and right-side address part points to the address of next node.




## Important Points to be Remembered

- ***In double linked list, the first node must be always pointed by head.***
- ***Always the previous field of the first node must be NULL.***
- ***Always the next field of the last node must be NULL.***

## Operations on Double Linked List

In a double linked list, we perform the following operations...

1) Insertion
2) Deletion
3) Display
4) Search
5) Sorting

### Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1) Inserting at Beginning of the list
2) Inserting at End of the list
3) Inserting at Specific location in the list

#### *1) Inserting at Beginning of the list*

We can use the following steps to insert a new node at beginning of the double linked list...

**Step 1** - Create a **newNode** with given value and **newNode → previous** as **NULL**.

**Step 2** - Check whether list is Empty **(head == NULL)**

**Step 3** - If it is Empty then, assign **NULL** to **newNode → next** and **newNode** to **head**.

**Step 4** - If it is not Empty then, assign **head** to **newNode → next** and **newNode** to **head**.

## *2) Inserting at End of the list*

We can use the following steps to insert a new node at end of the double linked list...

**Step 1** - Create a **newNode** with given value and **newNode → next as NULL.**

**Step 2** - Check whether list is Empty **(head == NULL)**

**Step 3** - If it is Empty, then assign NULL to **newNode → previous** and **newNode** to **head**.

**Step 4** - If it is not Empty, then, define a **node** pointer **ptr** and initialize with **head**.

**Step 5** - Keep moving the **ptr** to its next node until it reaches to the last node in the list (until **ptr → next** is equal to **NULL**).

**Step 6** - Assign **newNode** to **ptr→ next** and **ptr** to **newNode → previous**.

## *3) Inserting at Specific location in the list (After a Node)*

We can use the following steps to insert a new node after a node in the double linked list...

**Step 1** - Create a **newNode** with given value.

**Step 2** - Check whether list is Empty (head == NULL)

**Step 3** - If it is Empty then, assign NULL to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.

**Step 4** - If it is not Empty then, define two node pointers **ptr** & **temp** and initialize **ptr** with **head**.

**Step 5** - Keep moving the **ptr** to its next node until it reaches to the node after which we want to insert the **newNode** (until **ptr → data** is equal to location, here location is the node value after which we want to insert the **newNode**).

**Step 6** - Every time check whether **ptr** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **ptr** to **next node**.

**Step 7** - Assign **ptr → next** to **temp**, **newNode** to **ptr → next**, **ptr** to **newNode → previous**, **temp** to **newNode → next** and **newNode** to **temp → previous**.

## Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1) Deleting from Beginning of the list
2) Deleting from End of the list
3) Deleting a Specific Node

### *1) Deleting from Beginning of the list*

We can use the following steps to delete a node from beginning of the double linked list...

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is not Empty then, define a Node pointer **'ptr'** and initialize with **head**.

**Step 4 -** Check whether list is having only one node (**ptr → previous** is equal to **ptr → next**)

**Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **ptr** (Setting **Empty** list conditions)

**Step 6 -** If it is **FALSE**, then assign **ptr → next** to **head**, **NULL** to **head → previous** and delete **ptr**.

### *2) Deleting from End of the list*

We can use the following steps to delete a node from end of the double linked list...

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is not Empty then, define a Node pointer **'ptr'** and initialize with **head**.

**Step 4 -** Check whether list has only one Node (**ptr → previous** and **ptr → next** both are **NULL**)

**Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **ptr**. And terminate from the function. (Setting **Empty** list condition)

**Step 6 -** If it is **FALSE**, then keep moving **ptr** until it reaches to the last node in the list. (until **ptr → next** is equal to **NULL**)

**Step 7 -** Assign **NULL** to **ptr→ previous → next** and delete **ptr**.

### *3) Deleting a Specific Node from the list*

We can use the following steps to delete a specific node from the double linked list...

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is not Empty, then define a Node pointer **'ptr'** and initialize with **head**.

**Step 4 -** Keep moving the **ptr** until it reaches to the exact node to be deleted or to the last node.

**Step 5 -** If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the function.

**Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

**Step 7 -** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **ptr** (**free(ptr)**).

**Step 8 -** If list contains multiple nodes, then check whether **ptr** is the first node in the list (**ptr == head**).

**Step 9 -** If **ptr** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **ptr**.

**Step 10 -** If **ptr** is not the first node, then check whether it is the last node in the list (**ptr → next == NULL**).

**Step 11 -** If **ptr** is the last node then set **ptr** of **previous** of **next** to **NULL** (**ptr → previous → next = NULL**) and delete **ptr** (**free(ptr**)).

**Step 12 -** If **ptr** is not the first node and not the last node, then set **ptr** of **previous** of **next** to **ptr** of **next** (**ptr → previous → next = ptr → next**), **ptr** of **next** of **previous** to **ptr** of **previous** (**ptr → next → previous = ptr → previous**) and delete **ptr** (**free(ptr)**).

## Display

We can use the following steps to display the elements of a double linked list...

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.

**Step 3 -** If it is not Empty, then define a Node pointer **'ptr'** and initialize with **head**.

**Step 4 -** Display **'NULL <--- '**.

**Step 5 -** Keep displaying **ptr → data** with an arrow (**<===>**) until **ptr** reaches to the last node

**Step 6 -** Finally, display **ptr → data** with arrow pointing to **NULL** (**ptr → data ---> NULL**).

## Program for Doubly Linked List Operations

```
#include <stdio.h>
struct node
{
    int data;
    struct node *pre, *next;
};
struct node *head = NULL;
void insert(int);
void insertAtBeg(int);
void insertAtLoc(int, int);
void delete ();
void deleteBeg();
void deleteloc(int);
void display();
int countNode();
void main()
{
    int ch, val, loc;
    while (1)
    {
        printf("\nEnter your choice:\n\t1. Insert At Beginning\t\t2. Insert At S
pecific Location\t\t3. Insert At Last\n\t4. Delete From Front\t\t5. Delete From
Specific Location\t6. Delete From Last\n\t7. Count Nodes\t\t\t8. Display\t\t\t\t
0. Exit\n");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("Enter a value: ");
            scanf("%d", &val);
            insertAtBeg(val);
            break;
        case 2:
            printf("Enter value to insert: ");
            scanf("%d", &val);
            printf("Enter location: ");
            scanf("%d", &loc);
            insertAtLoc(val, loc);
            break;
        case 3:
            printf("Enter a value : ");
            scanf("%d", &val);
            insert(val);
            break;
        case 4:
            deleteBeg();
            break;
        case 5:
```

```
            printf("Enter the location of node you want to delete:  ");
            scanf("%d", &loc);
            deleteloc(loc);
            break;
        case 6:
            delete ();
            break;
        case 7:
            printf("\nTotal nodes in LL: %d", countNode());
            break;
        case 8:
            display();
            break;
        case 0:
            exit(0);
            break;

        default:
            printf("Enter a valid choice:::!!!");
            break;
        }
    }
}
void insertAtBeg(int a)
{
    struct node *ptr, *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = a;
    temp->next = NULL;
    if (head == NULL)
    {
        head = temp;
        temp->pre = NULL;
    }
    else
    {
        temp->next = head;
        head->pre = temp;
        head = temp;
    }
    display();
}

void insertAtLoc(int n, int l)
{
    struct node *temp, *ptr;
    int i;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = n;
    if (head == NULL)
```

```
    {
        temp->pre = temp->next = NULL;
        head = temp;
        printf("Value inserted!!!\n");
        display();
    }
    else if (l <= countNode())
    {
        if (l == 1)
        {
            insertAtBeg(n);
            printf("Value inserted!!!\n");
            display();
        }
        else
        {
            i = 2;
            ptr = head;
            while (i < l && ptr->next != NULL)
            {
                i++;
                ptr = ptr->next;
            }
            temp->pre = ptr;
            temp->next = ptr->next;
            ptr->next = temp;
        }
        printf("Value inserted!!!\n");
        display();
    }
    else if (l == (countNode()) + 1)
    {
        insert(n);
        printf("Value inserted!!!\n");
        display();
    }
    else
    {
        printf("Location not found!!!");
    }
}

void insert(int a)
{
    struct node *ptr, *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = a;
    temp->next = NULL;
    if (head == NULL)
    {
```

```
            temp->pre = NULL;
            head = temp;
        }
        else
        {
            ptr = head;
            while (ptr->next != NULL)
            {
                ptr = ptr->next;
            }
            ptr->next = temp;
            temp->pre = ptr;
        }
        display();
    }

    void deleteBeg()
    {
        struct node *ptr;
        ptr = head;
        if (head == NULL)
        {
            printf("nothing to delete !!");
        }
        else
        {
            if (head->next == head->pre) //head->next==NULL;
            {
                ptr = head;
                head = NULL;
                free(ptr);
                display();
            }
            else
            {
                ptr->next->pre = NULL;
                head = ptr->next;
                free(ptr);
                printf("Node deleted\n");
                display();
            }

        }
    }

    void deleteloc(int l)
    {
        struct node *ptr, *f;
        int i;
        if (head == NULL)
```

```
    {
        printf("Nothing to delete");
    }
    else
    {
        if (l == 1)
        {
            deleteBeg();
            display();
        }
        else if (l < countNode())
        {
            i = 2;
            ptr = head;
            while (i < l && ptr->next != NULL)
            {
                i++;
                ptr = ptr->next;
            }
            f = ptr->next;
            ptr->next=ptr->next->next;
            ptr->next->pre=f->pre;
            free(f);
            display();
        }
        else if(l==countNode())
        {
            delete();
        }
        else
        {
            printf("Desired Location not found");
        }
    }
}

void delete ()
{
    struct node *ptr, *f;
    ptr = head;

    if (head == NULL)
    {
        printf("Nothing to delete!!");
    }
    else
    {
        while (ptr->next != NULL)
        {
            ptr = ptr->next;
```

```
        }
        f = ptr;
        ptr->pre->next = NULL;
        free(f);
        printf("Node Deleted!!!\n");
        display();
    }
}

int countNode()
{
    struct node *ptr;
    int x = 0;
    ptr = head;
    while (ptr != NULL)
    {
        ptr = ptr->next;
        x++;
    }
    return x;
}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head==NULL)
    {
        printf("Nothing to display");
    }
    else
    {
        printf("Showing the elements of LL:-\n\t");
        while(ptr!=NULL)
        {
            printf("%d\t",ptr->data);
            ptr=ptr->next;
        }
    }
}
```

**Output:**

```
Enter your choice:-
     1. Insert at front         2. Insert at specific position       3. Insert at last
     4. Delete from Beginning   5. Delete from specific location     6. Delete from Last
     7. Count nodes             8. Display                           0. Exit

7

Total nodes in LL: 1
Enter your choice:-
     1. Insert at front         2. Insert at specific position       3. Insert at last
     4. Delete from Beginning   5. Delete from specific location     6. Delete from Last
     7. Count nodes             8. Display                           0. Exit
0

Enter your choice:-
     1. Insert at front         2. Insert at specific position       3. Insert at last
     4. Delete from Beginning   5. Delete from specific location     6. Delete from Last
     7. Count nodes             8. Display                           0. Exit

1
Enter Data:  12
Inserted !!
Showing the elements of LL:-
        12

Enter your choice:-
     1. Insert at front         2. Insert at specific position       3. Insert at last
     4. Delete from Beginning   5. Delete from specific location     6. Delete from Last
     7. Count nodes             8. Display                           0. Exit

2
Enter the value you want to Insert:  34
Enter location to insert: 2
Inserted!!
Showing the elements of LL:-
        12      34
Enter your choice:-
     1. Insert at front         2. Insert at specific position       3. Insert at last
     4. Delete from Beginning   5. Delete from specific location     6. Delete from Last
     7. Count nodes             8. Display                           0. Exit

4
Node Deleted
Showing the elements of LL:-
        34
```

## 3) Circular Linked List

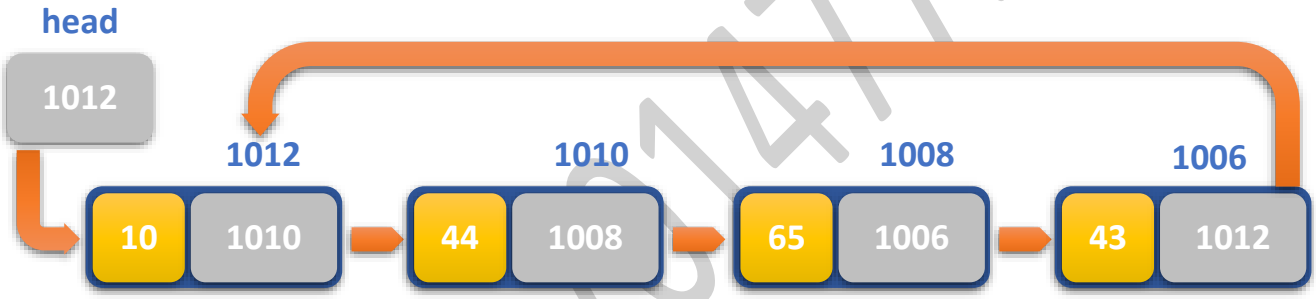
In single linked list, every node point to its next node in the sequence and the last node points NULL. But in circular linked list, every node point to its next node in the sequence but the last node points to the first node in the list.

*A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.*

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

### Representation of Circular Linked List

In Circular linked list the last node holds the address of the first node instead of NULL. So it creates a circular path in the linked-list.




### Operations on Circular Linked List

In a circular linked list, we perform the following operations...

1) Insertion
2) Deletion
3) Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

**Step 1 -** Include all the **header files** which are used in the program.

**Step 2 -** Declare all the **user defined** functions.

**Step 3 -** Define a **Node** structure with two members **data** and **next**

**Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.

**Step 5 -** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

## Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1) Inserting at Beginning of the list
2) Inserting at End of the list
3) Inserting at Specific location in the list

### *1) Inserting at Beginning of the list*

We can use the following steps to insert a new node at beginning of the circular linked list...

**Step 1 -** Create a **newNode** with given value.

**Step 2 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .

**Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

**Step 5 -** Keep moving the '**ptr**' to its next node until it reaches to the last node (until '**ptr → next** == **head**').

**Step 6 -** Set '**newNode → next** =**head**', '**head** = **newNode**' and '**ptr → next** = **head**'.

### *2) Inserting at End of the list*

We can use the following steps to insert a new node at end of the circular linked list...

**Step 1 -** Create a **newNode** with given value.

**Step 2 -** Check whether list is **Empty** (**head** == **NULL**).

**Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.

**Step 4 -** If it is **Not Empty** then, define a node pointer **ptr** and initialize with **head**.

**Step 5 -** Keep moving the **ptr** to its next node until it reaches to the last node in the list (until **ptr → next** == **head**).

**Step 6 -** Set **ptr → next** = **newNode** and **newNode → next** = **head**.

### *3) Inserting at Specific location in the list (After a Node)*

We can use the following steps to insert a new node after a node in the circular linked list...

**Step 1 -** Create a **newNode** with given value.

**Step 2 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.

**Step 4 -** If it is **Not Empty** then, define a node pointer **ptr** and initialize with **head**.

**Step 5 -** Keep moving the **ptr** to its next node until it reaches to the node after which we want to insert the newNode (until **ptr → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

**Step 6 -** Every time check whether **ptr** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **ptr** to next node.

**Step 7 -** If **ptr** is reached to the exact node after which we want to insert the newNode then check whether it is last node **(ptr→ next == head)**.

**Step 8 -** If **ptr** is last node then set **ptr → next** = **newNode** and **newNode → next** = **head**.

**Step 8 -** If **ptr** is not last node then set **newNode → next** = **ptr → next** and **ptr → next** = **newNode**.

## Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1) Deleting from Beginning of the list
2) Deleting from End of the list
3) Deleting a Specific Node

### *1) Deleting from Beginning of the list*

We can use the following steps to delete a node from beginning of the circular linked list...

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.

**Step 4 -** Check whether list is having only one node (**temp1 → next** == **head**)

**Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)

**Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head** )

**Step 7 -** Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

### *2) Deleting from End of the list*

We can use the following steps to delete a node from end of the circular linked list...

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.

**Step 4 -** Check whether list has only one Node (**temp1 → next** == **head**)

**Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

**Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)

**Step 7 -** Set **temp2 → next** = **head** and delete **temp1**.

### *3) Deleting a Specific Node from the list*

We can use the following steps to delete a specific node from the circular linked list...

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

**Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

**Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

**Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)

**Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

**Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).

**Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.

**Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).

**Step 11-** If **temp1** is last node then set **temp2→next** = **head** and delete **temp1** (**free(temp1)**).

**Step 12 -** If **temp1** is not first node and not last node then set **temp2→next** = **temp1→next** and delete **temp1** (**free(temp1)**).

## Display

We can use the following steps to display the elements of a circular linked list...

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.

**Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

**Step 4 -** Keep displaying **temp → data** with an arrow (--->) until **temp** reaches to the last node

**Step 5 -** Finally display **temp → data** with arrow pointing to **head → data**.

**Program for Circular Linked List Operations**

```
#include <stdio.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head = NULL;
void insertBeg(int a);
void insertAtLoc(int n, int l);
void insertAtLast(int n);
void deleteBeg();
void dltSpc(int l);
void deleteLast();
void display();
void main()
{
    int ch, val, loc;
    while (1)
    {
        printf("\n\nEnter your choice:-
\n\t1. Insert at front\t\t2. Insert at specific position\t\t3. Insert at last\n\
t4. Delete from Beginning\t5. Delete from specific location\t6. Delete from Last
\n\t7. Count nodes\t\t\t8. Display\t\t\t\t0.Exit\n\n");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("Enter Data:  ");
            scanf("%d", &val);
            insertBeg(val);
            break;
```

```
            case 2:
                printf("Enter the value you want to Insert:  ");
                scanf("%d", &val);
                printf("Enter location to insert: ");
                scanf("%d", &loc);
                insertAtLoc(val, loc);
                break;

            case 3:
                printf("Enter Data:  ");
                scanf("%d", &val);
                insertAtLast(val);
                break;

            case 4:
                deleteBeg();
                break;
            case 5:
                printf("Enter location to remove node from LL: ");
                scanf("%d", &loc);
                dltSpc(loc);
                break;

            case 6:
                deleteLast();
                break;

            case 7:
                printf("\nTotal nodes in LL: %d", countNode());
                break;

            case 8:
                display();
                break;

            case 0:
                exit(0);
                break;
            default:
                printf("Enter a valid choice !!");
                break;
            }
        }
}
void insertBeg(int a)
{
    struct node *temp, *ptr;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = a;
```

```
    if (head == NULL)
    {
        head = temp;
        temp->next = head;
    }
    else
    {
        ptr = head;
        while (ptr->next != head)
        {

            ptr = ptr->next;
        }
        temp->next = head;
        head = temp;
        ptr->next = head;
    }
    printf("Inserted !!\n");
    display();
}

void insertAtLoc(int n, int l)
{
    struct node *temp, *ptr;
    int i;
    if (head == NULL)
    {
        printf("Linked List is empty!!");
    }
    else
    {
        temp = (struct node *)malloc(sizeof(struct node));
        temp->data = n;
        if (l == 1)
        {
            insertBeg(n);
        }
        else
        {
            i = 2;
            ptr = head;
            while (i < l && ptr->next != head)
            {
                i++;
                ptr = ptr->next;
            }
            temp->next = ptr->next;
            ptr->next = temp;
            printf("Inserted!!\n");
            display();
```

```
        }
    }
}

void insertAtLast(int n)
{
    struct node *temp, *ptr;

    temp = (struct node *)malloc(sizeof(struct node));

    temp->data = n;

    if (head == NULL)
    {
        insertBeg(n);
    }

    else

    {
        ptr = head;
        while (ptr->next != head)
        {
            ptr = ptr->next;
        }
        ptr->next = temp;
        temp->next = head;
        printf("Inserted !!\n");
        display();
    }
}

void deleteBeg()
{
    struct node *ptr, *f;
    if (head == NULL)
    {
        printf("Nothing to delete");
    }
    else
    {
        if (head->next == head)
        {
            head = NULL;
            free(head);
        }
        else
        {
            f = ptr = head;
            while (ptr->next != head)
```

```
            {

                ptr = ptr->next;
            }
            ptr->next = f->next;
            head = f->next;
            free(f);
            printf("Node Deleted\n");
        }

        display();
    }
}

void dltSpc(int l)
{
    struct node *ptr, *f;
    int i;
    if (head == NULL)
    {
        printf("Nothing to delete!!!");
    }
    else
    {
        if (l == 1)
        {
            deleteBeg();
        }
        else if (l <= countNode())
        {
            i = 2;
            ptr = head;
            while (i < l && ptr->next != head)
            {
                i++;
                ptr = ptr->next;
            }

            f = ptr->next;
            ptr->next = ptr->next->next;
            free(f);
            printf("One Node deleted!!!!\n");
            display();
        }
        else
        {
            printf("Desired location not found!! Try again!!\n");
            display();
        }
    }
```

```
}

void deleteLast()
{
    struct node *ptr, *f;
    if (head == NULL)
    {
        printf("Nothing to delete!!!");
    }
    else
    {
        ptr = head;
        if (head->next == head)
        {
            f = head;
            head = NULL;
            free(f);
        }
        else
        {
            while (ptr->next->next != head)
            {
                ptr = ptr->next;
            }
            f = ptr->next;
            ptr->next = head;
            free(f);
        }
        printf("Node deleted from the last!!\n");
        display();
    }
}

int countNode()
{
    struct node *ptr;
    int x = 0;
    ptr = head;
    do
    {
        ptr = ptr->next;
        x++;
    } while (ptr != head);

    return x;
}

void display()
{
    struct node *ptr;
```

```
    ptr = head;
    if (head == NULL)
    {
        printf("Nothing to display");
    }
    else
    {
        printf("Showing the elements of LL:-\n\t");
        do
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        } while (ptr != head);
    }
}
```

**Output:**

```
Enter your choice:-
        1. Insert at front              2. Insert at specific position          3. Insert
at last
        4. Delete from Beginning        5. Delete from specific location        6. Delete
from Last
        7. Count nodes                  8. Display                              0.Exit

1
Enter Data:  23
Inserted !!
Showing the elements of LL:-
        23
Enter your choice:-
     1. Insert at front         2. Insert at specific position       3. Insert at last
     4. Delete from Beginning   5. Delete from specific location     6. Delete from Last
     7. Count nodes             8. Display                           0. Exit

3
Enter Data:  26
Inserted !!
Showing the elements of LL:-
        23      26

Enter your choice:-
     1. Insert at front         2. Insert at specific position       3. Insert at last
     4. Delete from Beginning   5. Delete from specific location     6. Delete from Last
     7. Count nodes             8. Display                           0. Exit
2
Enter the value you want to Insert:  33
Enter location to insert: 3
Inserted!!
Showing the elements of LL:-
        23      26      33

Enter your choice:-
     1. Insert at front         2. Insert at specific position       3. Insert at last
     4. Delete from Beginning   5. Delete from specific location     6. Delete from Last
     7. Count nodes             8. Display                           0. Exit

7

Total nodes in LL: 3
```

```
Enter your choice:-
     1. Insert at front          2. Insert at specific position       3. Insert at last
     4. Delete from Beginning    5. Delete from specific location     6. Delete from Last
     7. Count nodes              8. Display                           0. Exit
6
Node deleted from the last!!
Showing the elements of LL:-
        23      26

Enter your choice:-
     1. Insert at front          2. Insert at specific position       3. Insert at last
     4. Delete from Beginning    5. Delete from specific location     6. Delete from Last
     7. Count nodes              8. Display                           0. Exit
0
```

# Tree

In linear data structure data is organized in sequential order and in 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. A tree data structure can be defined as follows...

*Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.*

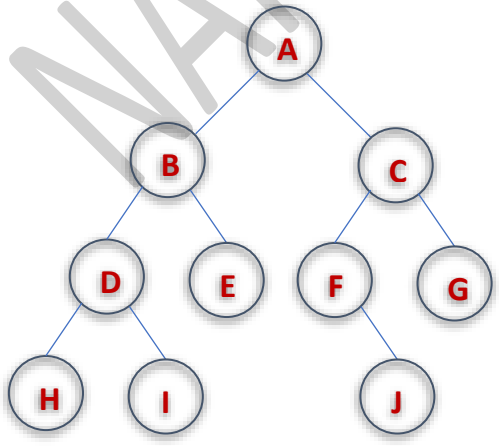
**OR**

*Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively.*

In 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.

In a tree, if we have **n** number of nodes then we can have a maximum of **n-1** number of links.

**Example**




**TREE with 10 nodes and 9 edges**

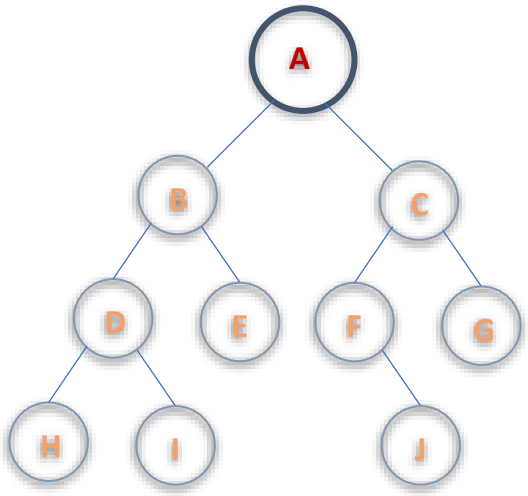
- **In any tree with '*n*' nodes there will be maximum of 'n-1' edges**
- **In a tree every individual element is called as 'NODE'**

## Terminology

In a tree data structure, we use the following terminology...

### 1) Root

In a tree data structure, the first node is called as Root Node. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

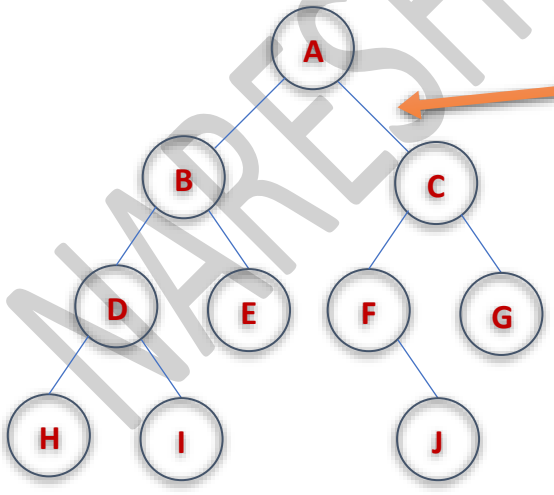



**Here 'A' is the 'root' node**

- **In any tree the first node is called as ROOT node**

### 2) Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

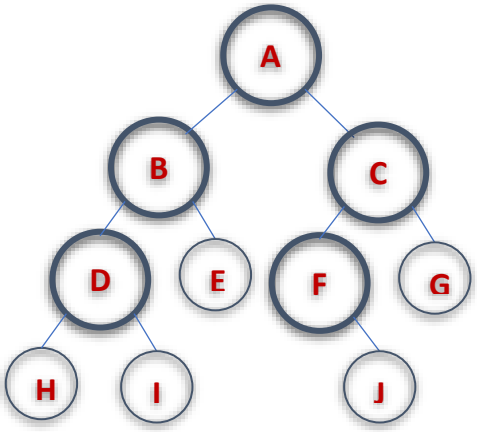



- **In any tree 'Edge' is a connecting link between two nodes**

### 3) Parent

In a tree data structure, the node which is a predecessor of any node is called as PARENT NODE.

In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "The node which has child / children".

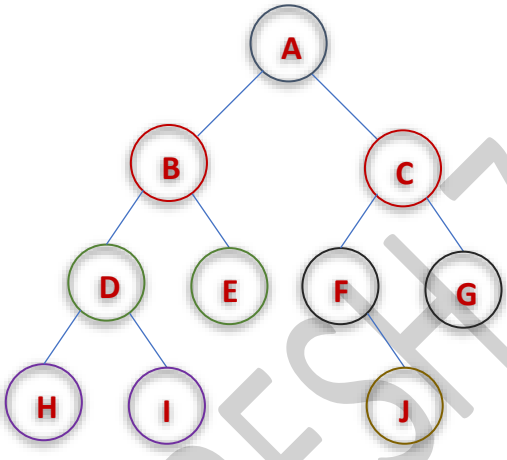



**Here A, B, C, D & F are Parent nodes**

- **In any tree the node which has child / children is called 'Parent'**
- **A node which is predecessor of any other node is called 'Parent'**

## 4) Child

In a tree data structure, the 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. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.




**Here B & C are children of A**

**Here D & E are children of B**

**Here F & G are children of C**

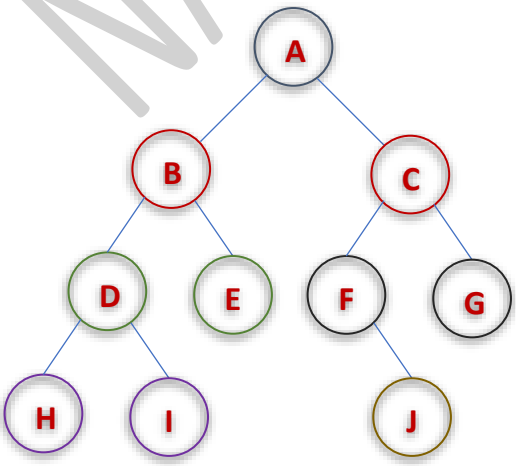
**Here H & I are children of D**

**Here J are children of F**

- **Descendent of any node is called as CHILD node**

## 5) Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with the same parent are called Sibling nodes.




**Here B & C are Siblings**

**Here D & E are Siblings**

**Here F & G are Siblings**

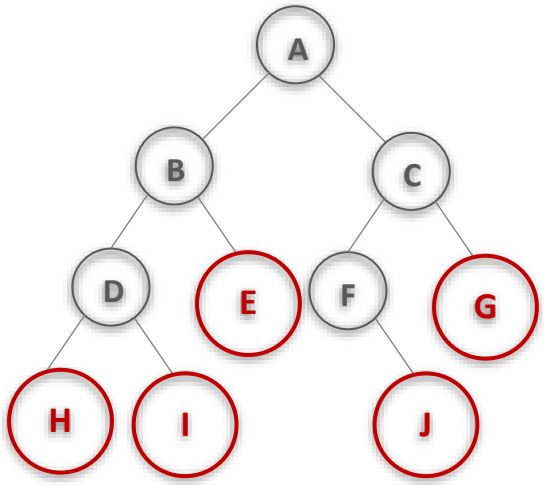
**Here H & I Siblings**

- **In any tree the nodes with same parent are called 'Siblings'**

## 6) Leaf

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child.

*In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, **leaf node** is also called as **'Terminal'** node.*



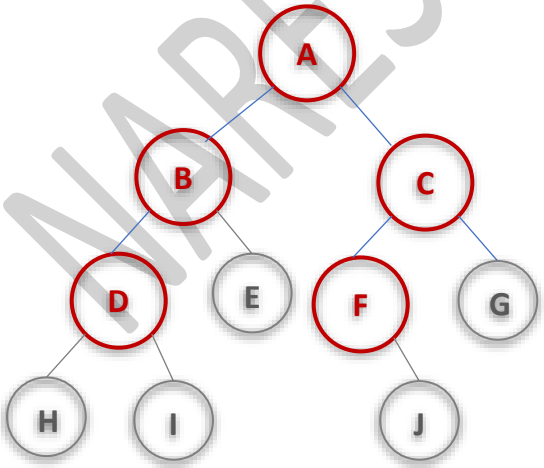

**Here E, G, H, I & J are Leaf nodes**

- **In any tree the nodes which does not have children is called 'Leaf'.**
- **A node without successor is called 'Leaf' node.**

## 7) Internal Nodes

In a tree data structure, the node which has at least one child is called as INTERNAL Node. In simple words, an internal node is a node with at least one child.

In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



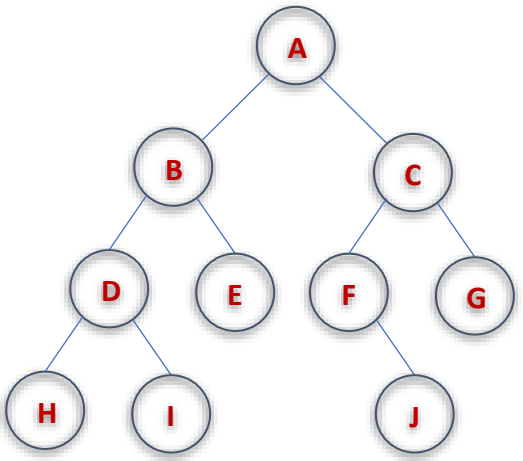

**Here A, B, C, D & F are Internal nodes**

- **In any tree the nodes which has atleast one child is called 'Internal' node.**
- **Every non-leaf node is called as 'Internal' node.**

## 8) Degree

In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has.

The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'.




**Here Degree of B is 2**

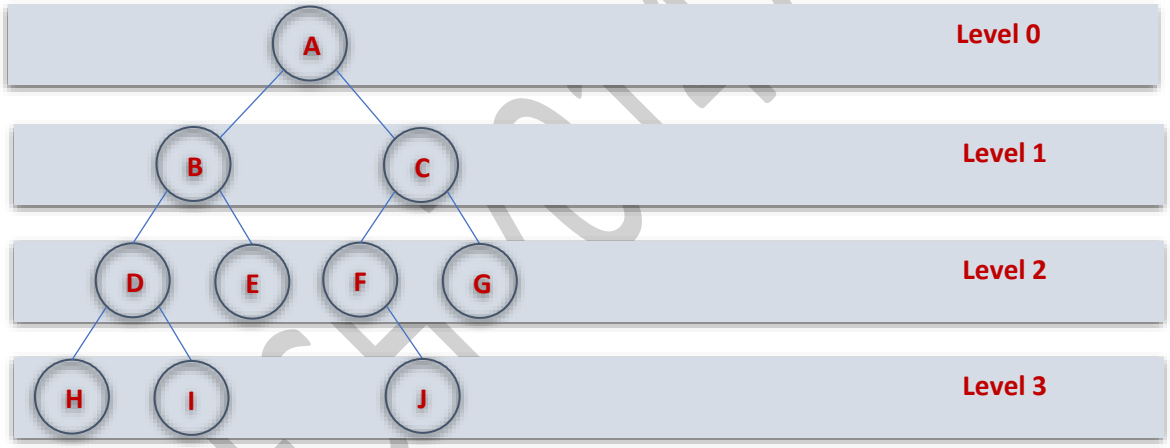
**Here Degree of A is 2**

**Here Degree of F is 1**

- **In any tree 'Degree' of a node is total number of children it has.**

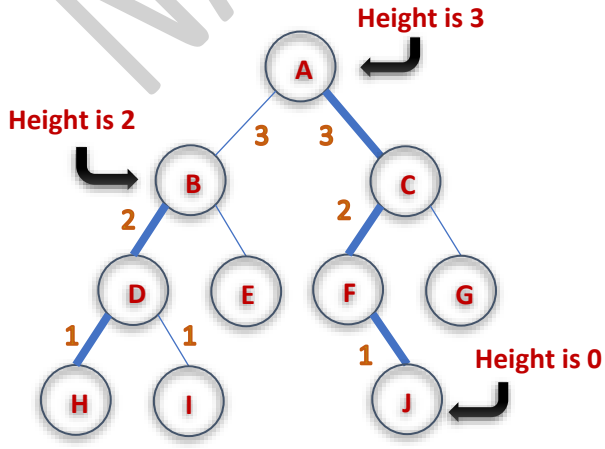
## 9) Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree 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 (Step).




## 10) Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

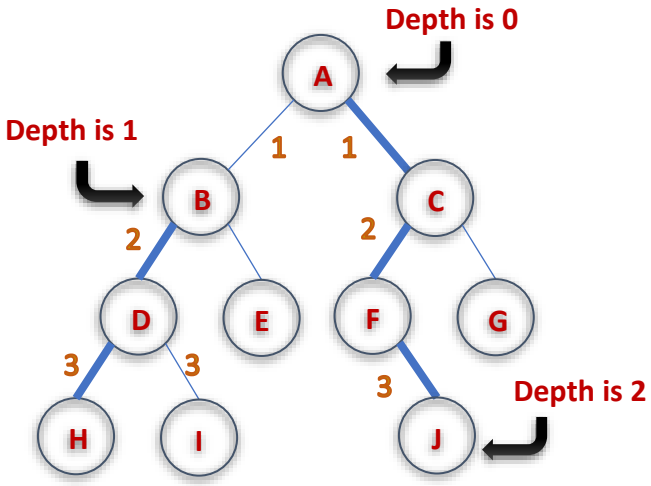



**Here Height of tree is 3**

- **In any tree 'Height of Node' is total number of edges from leaf to that node in longest path.**
- **In ant tree, 'Height of Tree' is the height of the root node.**

## 11) Depth

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

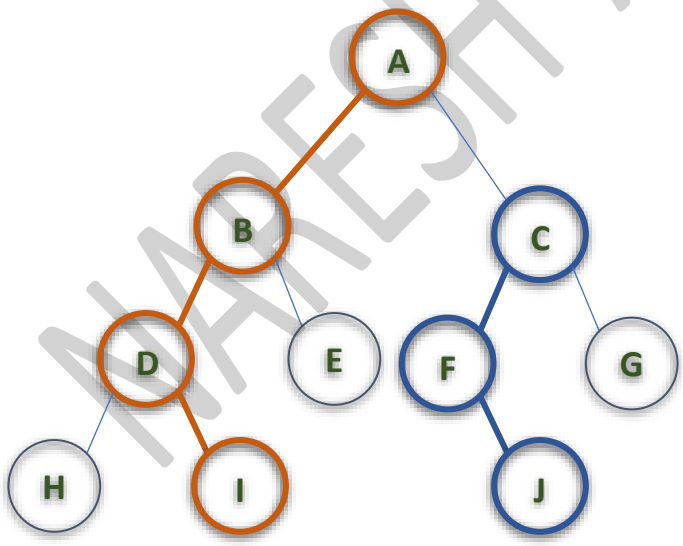



**Here Depth of tree is 3**

- **In any tree 'Depth of Node' is total number of edges from root to that node.**
- **In ant tree, 'Depth of Tree' is the total number of edges from root to leaf in longest path.**

## 12) Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.




**Here 'Path' between A & I is**

**A – B – D – I**

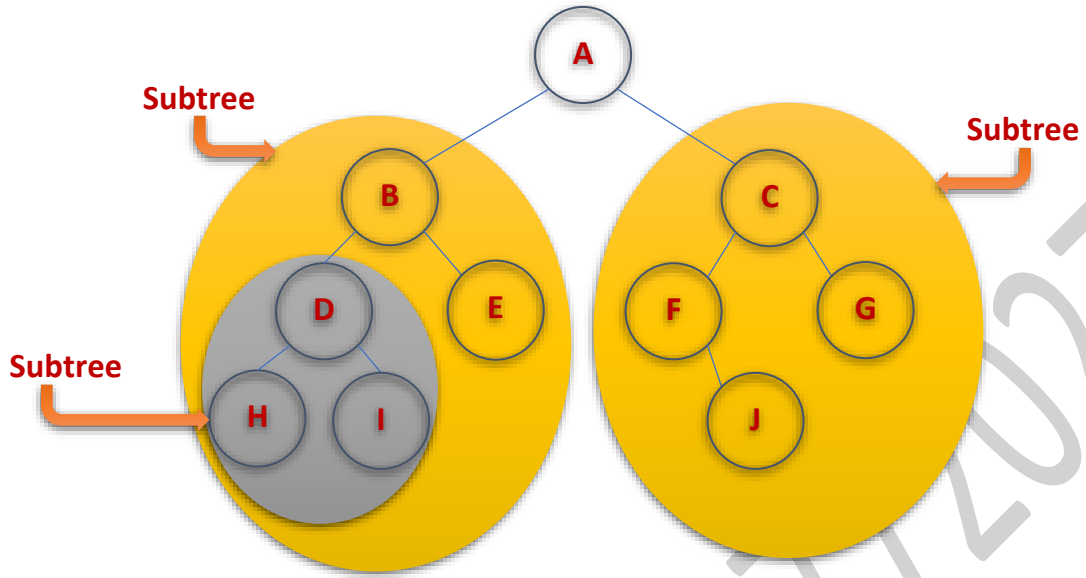
**Here 'Path' between C & J is**

**C – F – J**

- **In any tree 'Path' is a sequence of nodes and edges between two nodes.**

## 13) Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

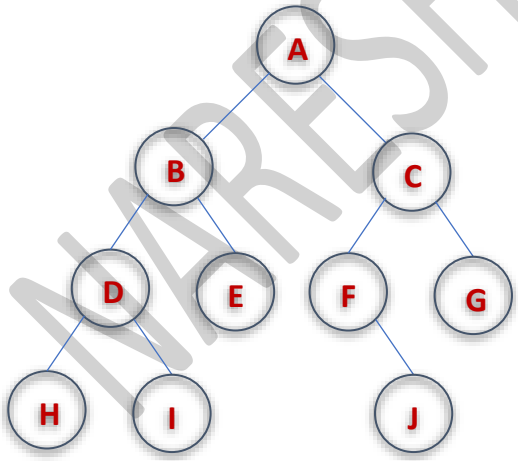



# Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1) **List Representation**
2) **Left Child - Right Sibling Representation**

Consider the following tree...




**Tree with 10 nodes and 9 edges.**

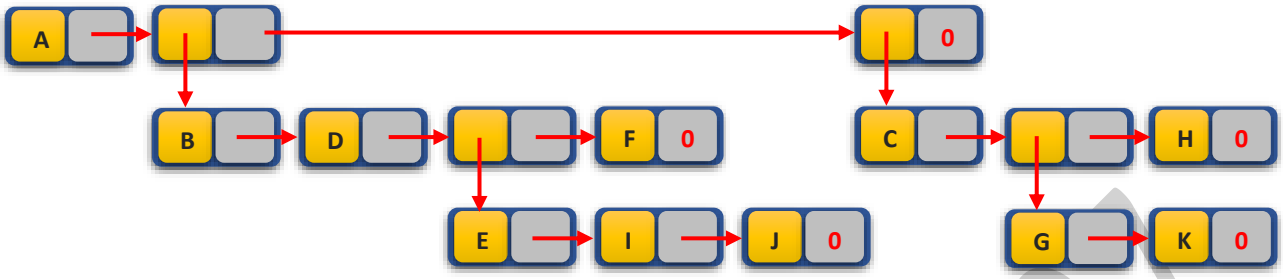
- **In any tree with 'N' node there will be maximum of 'N-1' edges.**
- **In any tree every individual element is called as 'Node'.**

## 1) List Representation

In this representation, we use two types of nodes one for representing the node with data called 'data node' and another for representing only references called 'reference node'. We start with a 'data node' from the root node in the tree. Then it is linked to an internal node

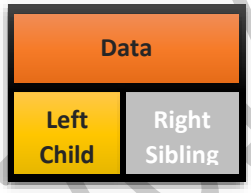
through a 'reference node' which is further linked to any other node directly. This process repeats for all the nodes in the tree.

The above example tree can be represented using List representation as follows...



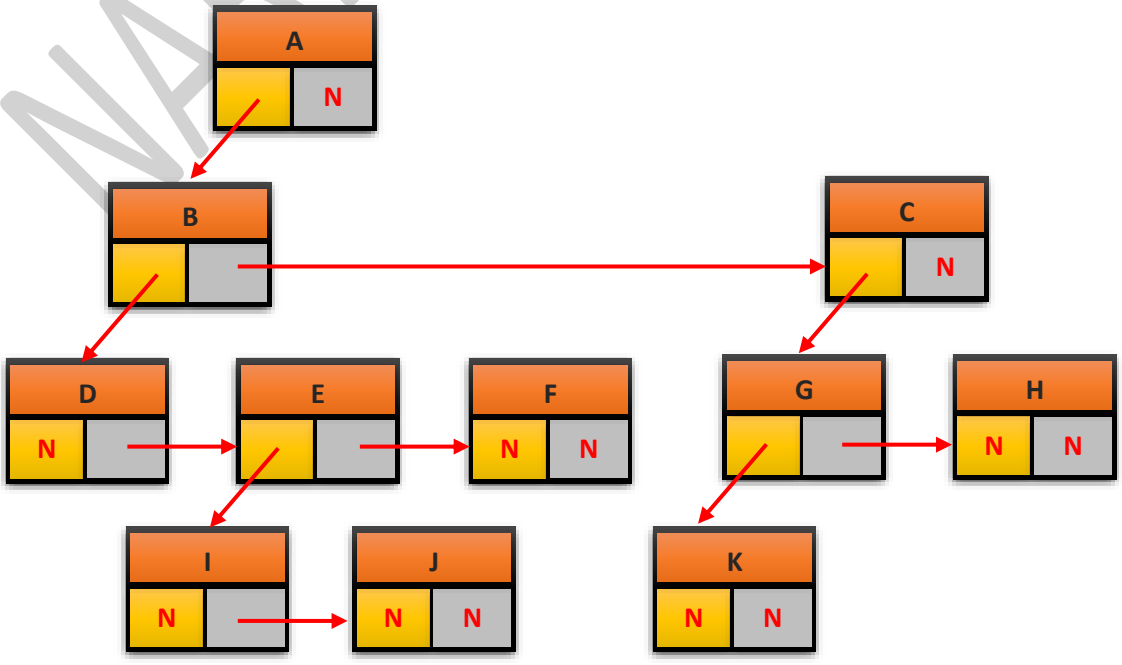

## 2) Left Child - Right Sibling Representation

In this representation, we use a list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...




In this representation, every node's data field stores the actual value of that node. If that node has left a child, then left reference field stores the address of that left child node otherwise stores NULL. If that node has the right sibling, then right reference field stores the address of right sibling node otherwise stores NULL.

The above example tree can be represented using Left Child - Right Sibling representation as follows...



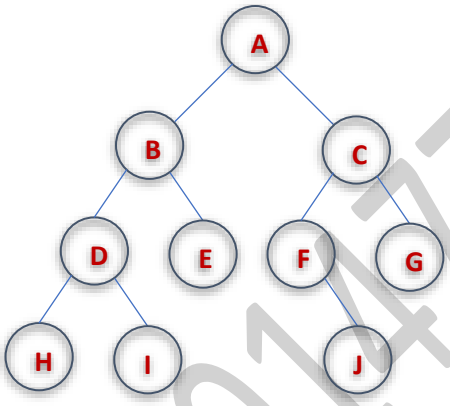
## Binary Tree Data structure

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**. One is known as a left child and the other is known as right child.

*A tree in which every node can have a maximum of two children is called Binary Tree.*

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

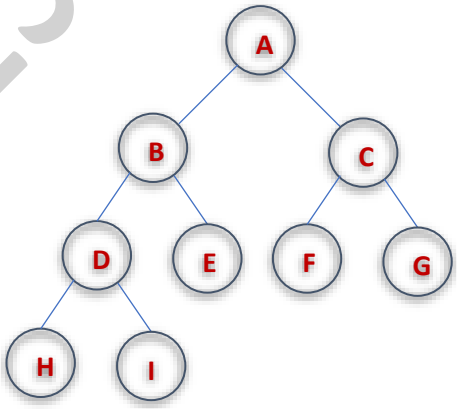
**Example**




There are different types of binary trees and they are...

### 1) Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

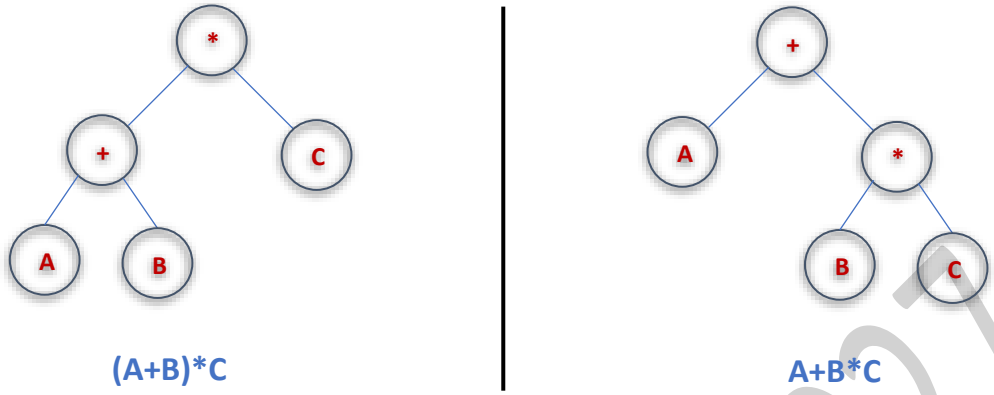



*A binary tree in which every node has either two or zero number of children is called* ***Strictly Binary Tree.***

*Strictly binary tree is also called as* ***Full Binary Tree*** *or* ***Proper Binary Tree*** *or* ***2-Tree.***

Strictly binary tree data structure is used to represent mathematical expressions.

**Example**




(A+B)*C A+B*C

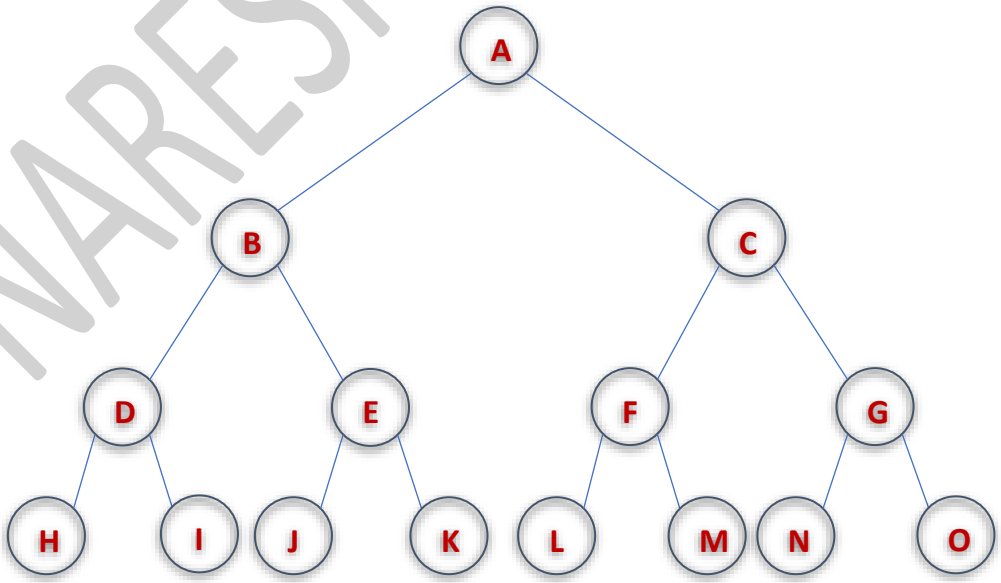
## 2) Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be $2^{level}$ number of nodes. For example: at level 2 there must be $2^2 = 4$ nodes and at level 3

*A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.*

there must be $2^3 = 8$ nodes.

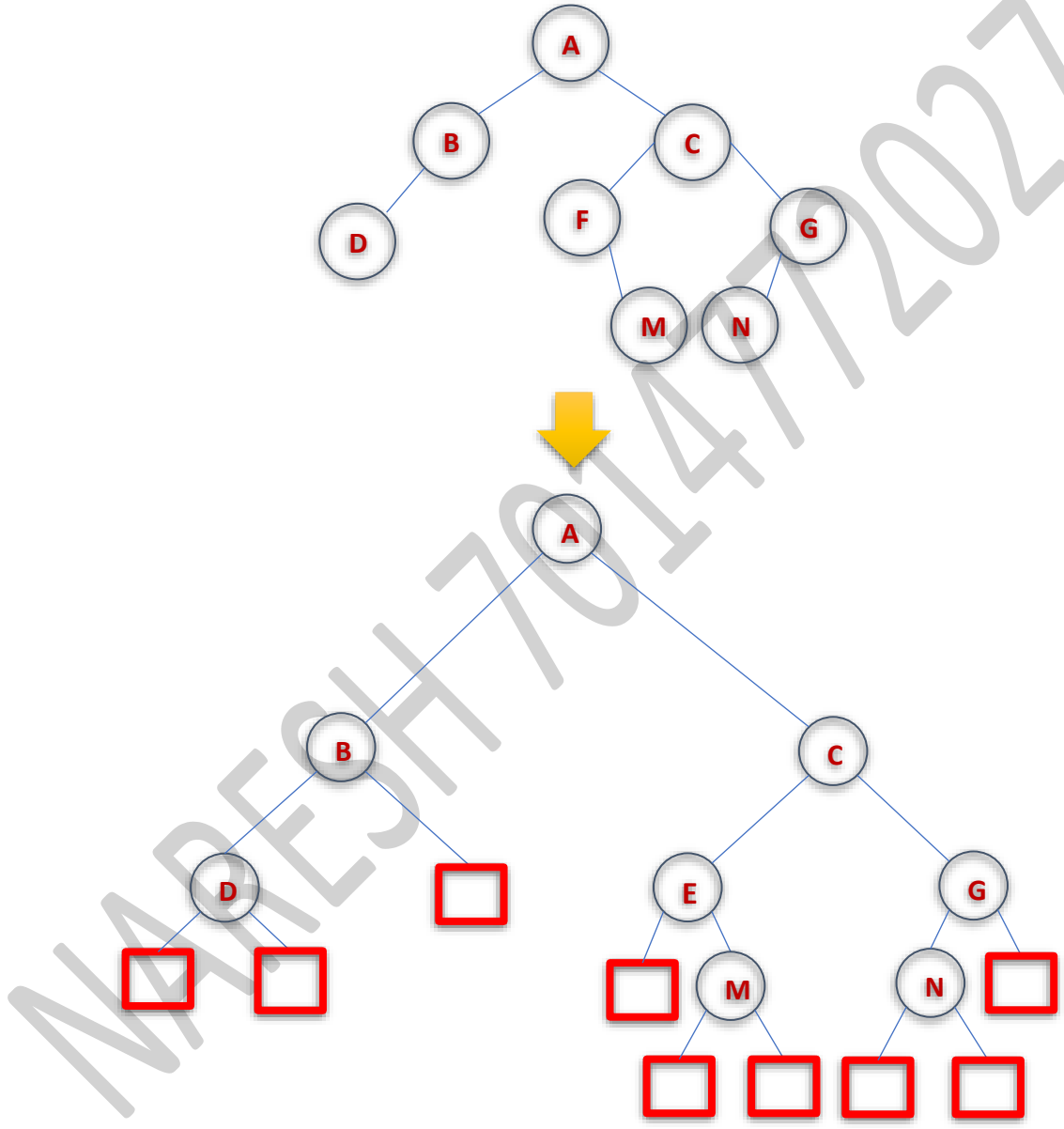
Complete binary tree is also called as **Perfect Binary Tree**



## 3) Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

*The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.*






In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink color).

## Threaded Binary Trees

A binary tree can be represented using array representation or linked list representation. When a binary tree is represented using linked list representation, the reference part of the node which doesn't have a child is filled with a NULL pointer. In any binary tree linked list

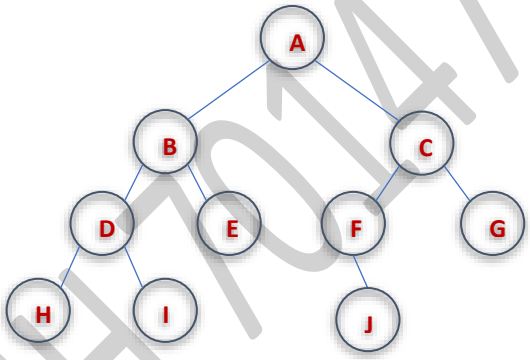
representation, there is a number of NULL pointers than actual pointers. Generally, in any binary tree linked list representation, if there are 2N number of reference fields, then N+1

*Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.*

number of reference fields are filled with NULL (N+1 is NULL out of 2N). This NULL pointer does not play any role except indicating that there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "Threaded Binary Tree", which makes use of NULL pointers to improve its traversal process. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as threads.
If there is no in-order predecessor or in-order successor, then it points to the root node.

Consider the following binary tree...



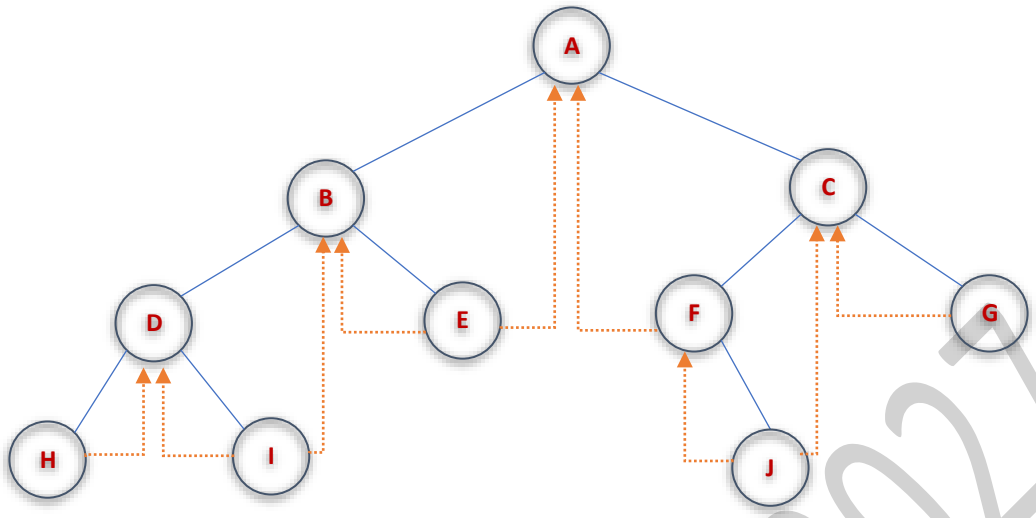

To convert the above example binary tree into a threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

**H - D - I - B - E - A - F - J - C - G**

When we represent the above binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL. This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes H, I, E, J and G right child pointers are NULL. These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

Above example binary tree is converted into threaded binary tree as follows.



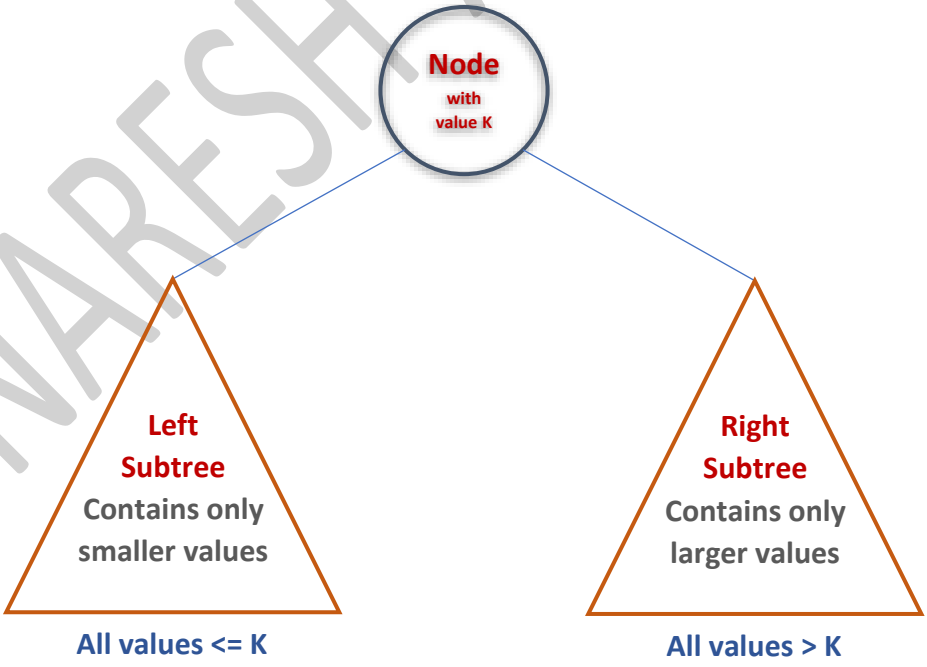

In the above figure, threads are indicated with dotted links.

## Binary Search Tree

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.

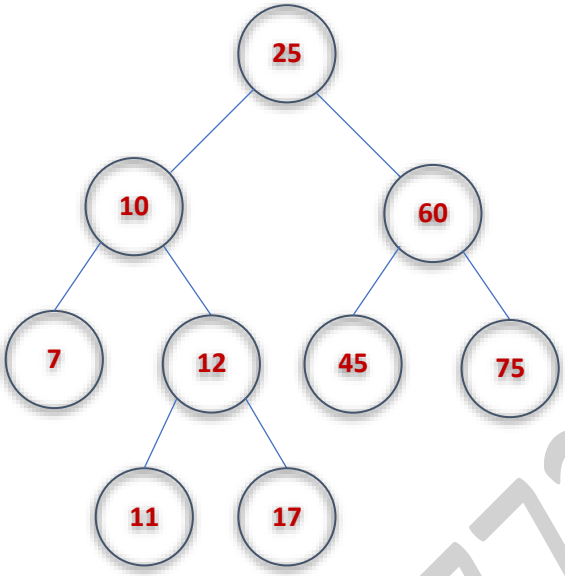
A binary tree has the following time complexities...

- Search Operation - O(n)
- Insertion Operation - O(1)
- Deletion Operation - O(n)




*Every binary search tree is a binary tree but every binary tree need not to be binary search tree.*

Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root. Similarly, value of all the




nodes in the right sub-tree is greater than or equal to the value of the root. This rule will be recursively applied to all the left and right sub-trees of the root.

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 25 doesn't contain any value greater than or equal to 25 in its left sub-tree and it also doesn't contain any value less than 25 in its right sub-tree.

## Advantages of using binary search tree

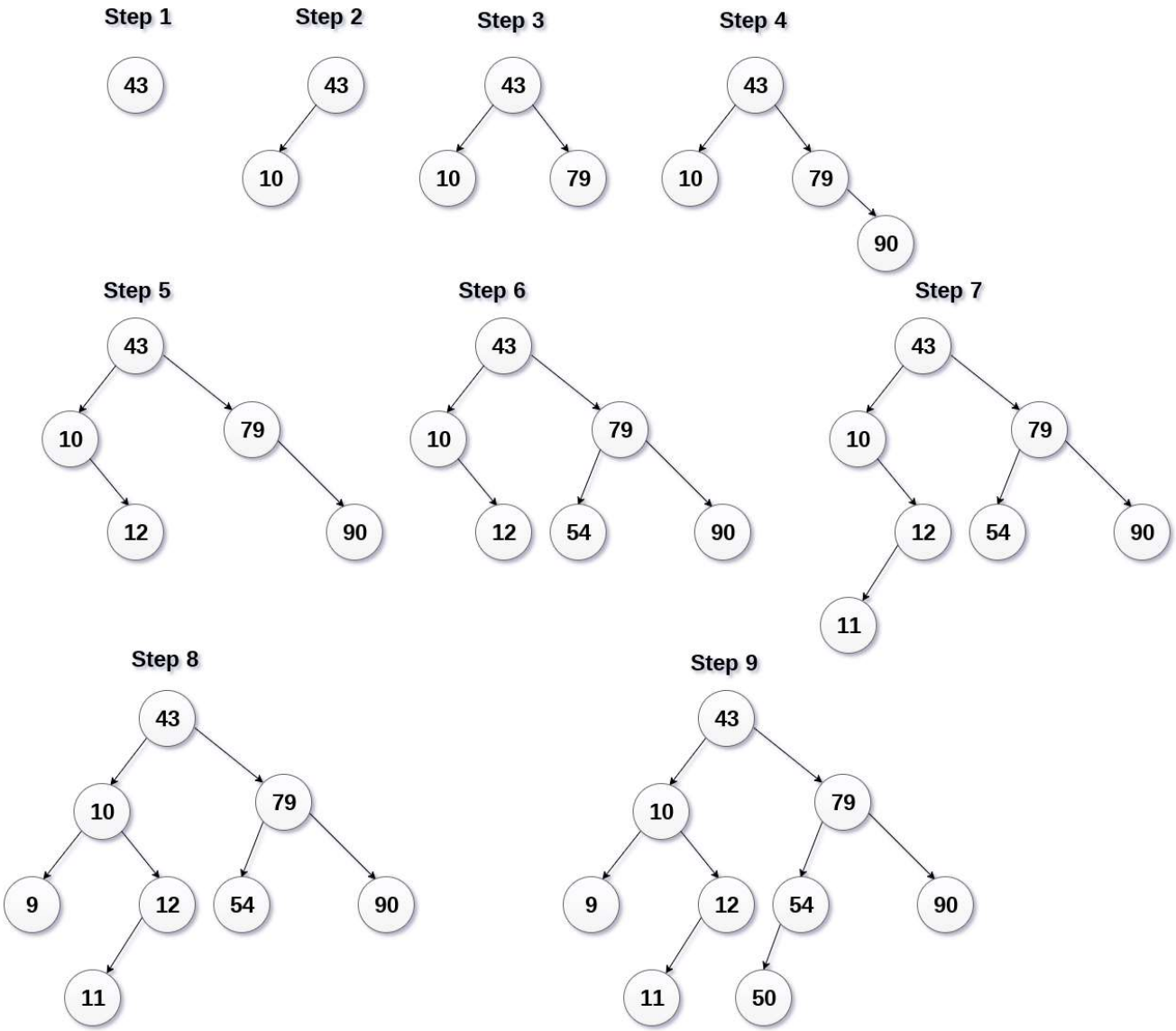
- Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
- The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o(log2n) time. In worst case, the time it takes to search an element is 0(n).
- It also speeds up the insertion and deletion operations as compare to that in array and linked list.

**Example: Create the binary search tree using the following data elements.**

**43, 10, 79, 90, 12, 54, 11, 9, 50**

1) Insert 43 into the tree as the root of the tree.
2) Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3) Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown below.




**Binary search Tree Creation**

## Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1) Search
2) Insertion
3) Deletion

### 1) Search

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

**Step 1 -** Read the search element from the user.

**Step 2 -** Compare the search element with the value of root node in the tree.

**Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function

**Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.

**Step 5 -** If search element is smaller, then continue the search process in left subtree.

**Step 6-** If search element is larger, then continue the search process in right subtree.

**Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node

**Step 8 -** If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

**Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

## 2) Insertion

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. 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.

## 3) Deletion

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...

1) **Case 1: Deleting a Leaf node (A node with no children)**
2) **Case 2: Deleting a node with one child**
3) **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.

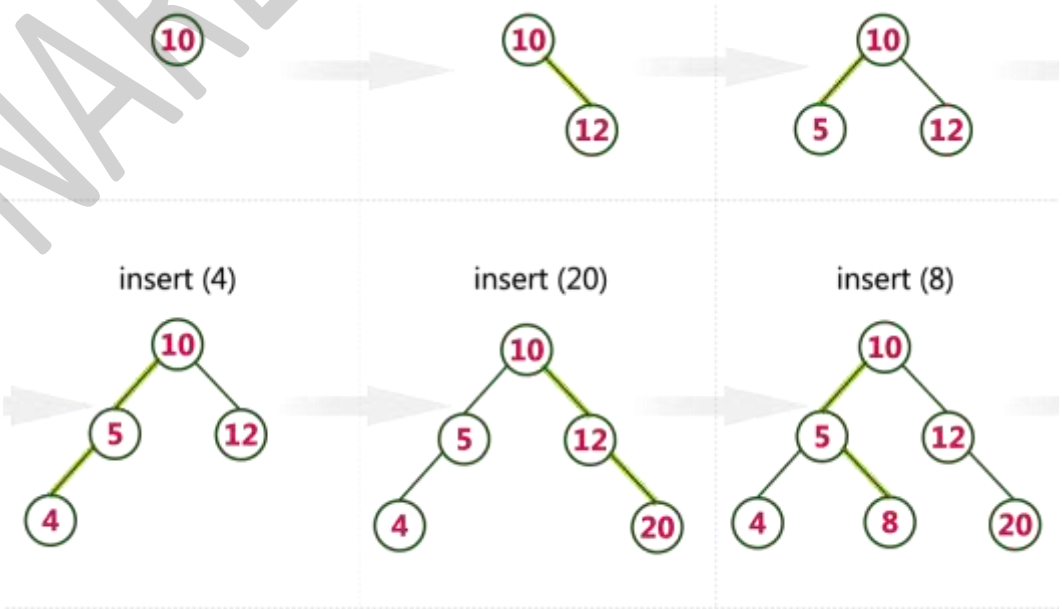
**Step 6 -** If it comes to case 2, then delete using case 2 logic.

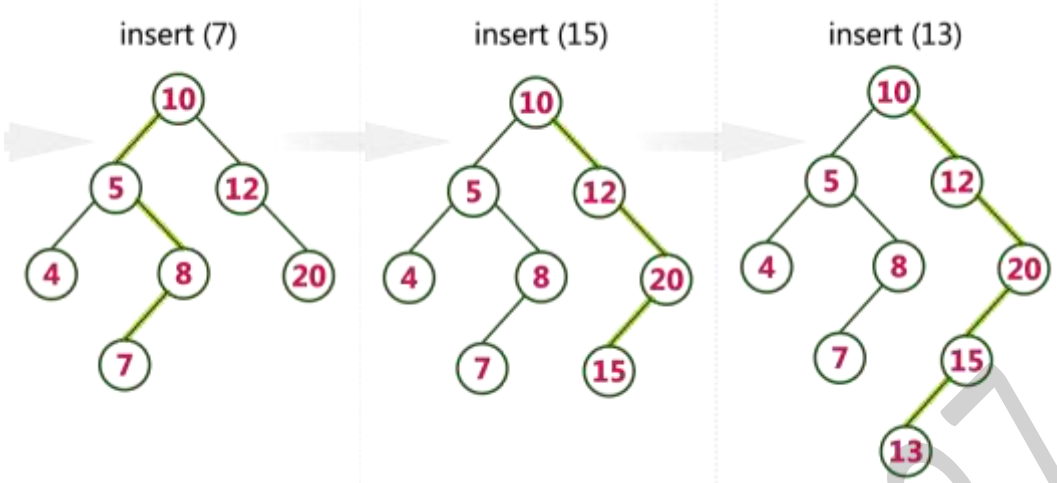
**Step 7 -** Repeat the same process until the node is deleted from the tree.

**Example: 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...






**Example:**

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
void inorder(struct node *root)
{
    if (root)
    {
        inorder(root->left);
        printf("  %d", root->data);
        inorder(root->right);
    }
}
void main()
{
    int n, i;
    struct node *p, *q, *root;
    printf("Enter the number of nodes to be insert: ");
    scanf("%d", &n);
    printf("\nPlease enter the numbers to be insert: ");
    for (i = 0; i < n; i++)
    {
        p = (struct node *)malloc(sizeof(struct node));
        scanf("%d", &p->data);
        p->left = NULL;
        p->right = NULL;
        if (i == 0)
        {
            root = p; // root always point to the root node
        }
        else
```

```
        {
            q = root; // q is used to traverse the tree
            while (1)
            {
                if (p->data > q->data)
                {
                    if (q->right == NULL)
                    {
                        q->right = p;
                        break;
                    }
                    else
                        q = q->right;
                }
                else
                {
                    if (q->left == NULL)
                    {
                        q->left = p;
                        break;
                    }
                    else
                        q = q->left;
                }
            }
        }
    }
    printf("\nBinary Search Tree nodes in In-order Traversal: ");
    inorder(root);
    printf("\n");
}
```

**Output:**

```
Enter the number of nodes to be insert: 5
Please enter the numbers to be insert: 98
7
12
5
76
Binary Search Tree nodes in In-order Traversal:   5  7  12  76  98
```

## AVL Tree

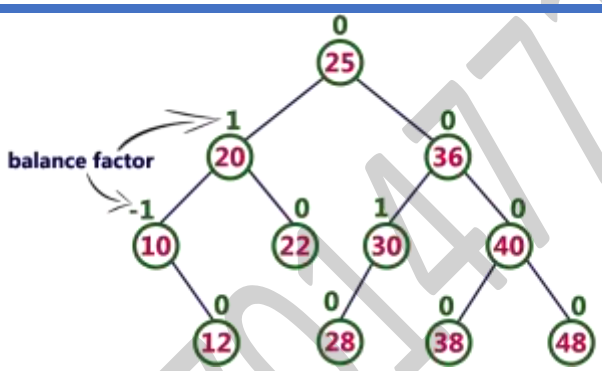
AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of

every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**. The AVL tree was introduced in the year 1962 by G.M. **Adelson-Velsky** and **E.M. Landis**. An AVL tree is defined as follows...

*An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.*

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we calculate as follows...

*Balance factor = heightOfLeftSubtree - heightOfRightSubtree*




The above tree is a binary search tree and every node is satisfying balance factor condition. So, this tree is said to be an AVL tree.

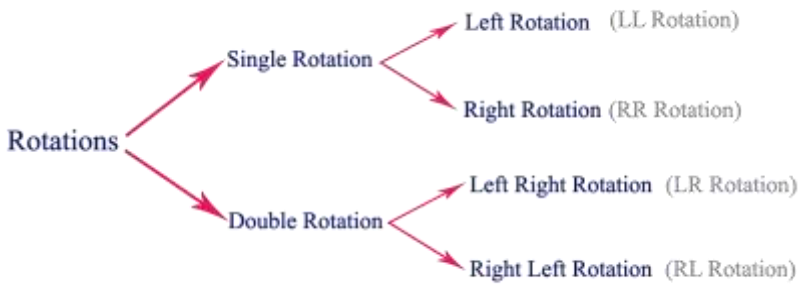
*Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.*

## AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation, we use **rotation** operations to make the tree balanced. Rotation operations are used to make the tree balanced.

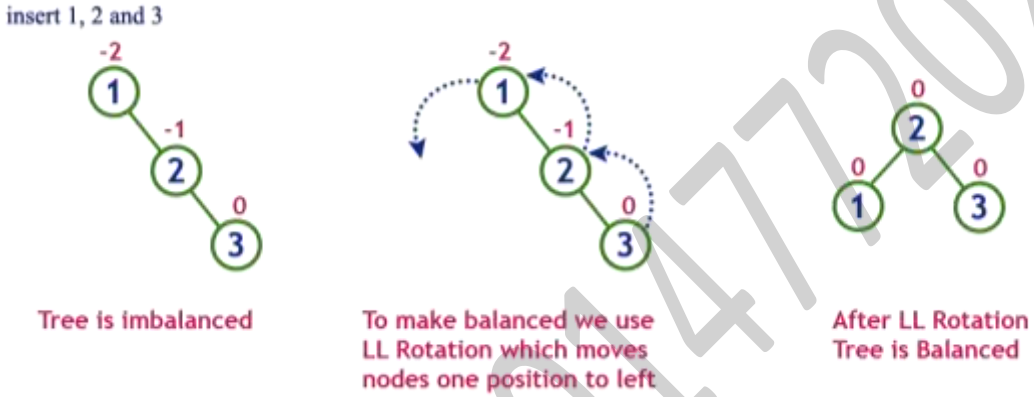
*Rotation is the process of moving nodes either to left or to right to make the tree balanced.*

There are **four** rotations and they are classified into **two** types.



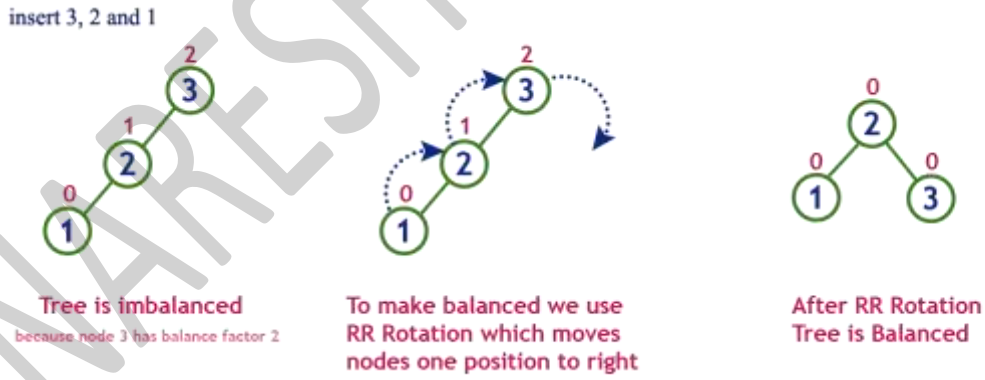

## Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



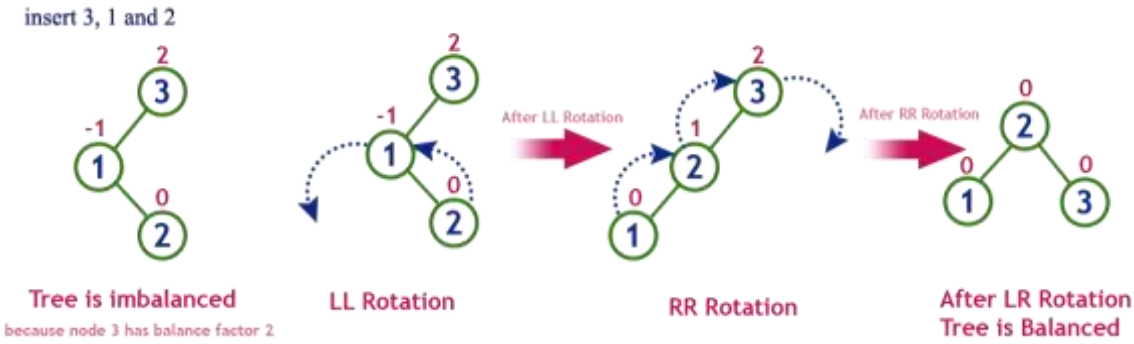

## Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



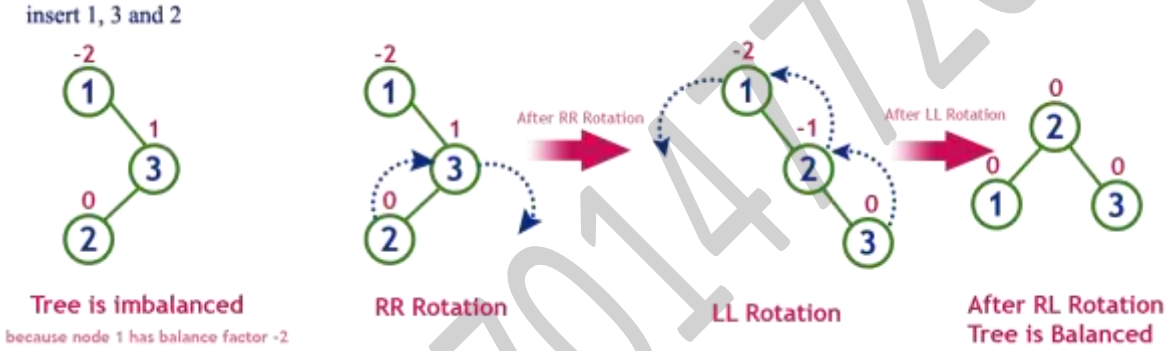

## Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...




## Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...




## Operations on an AVL Tree

The following operations are performed on AVL tree...

- Search
- Insertion
- Deletion

## Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

**Step 1** - Read the search element from the user.

**Step 2 -** Compare the search element with the value of root node in the tree.

**Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function

**Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.

**Step 5 -** If search element is smaller, then continue the search process in left subtree.

**Step 6 -** If search element is larger, then continue the search process in right subtree.

**Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node.

**Step 8 -** If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.

**Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

## Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

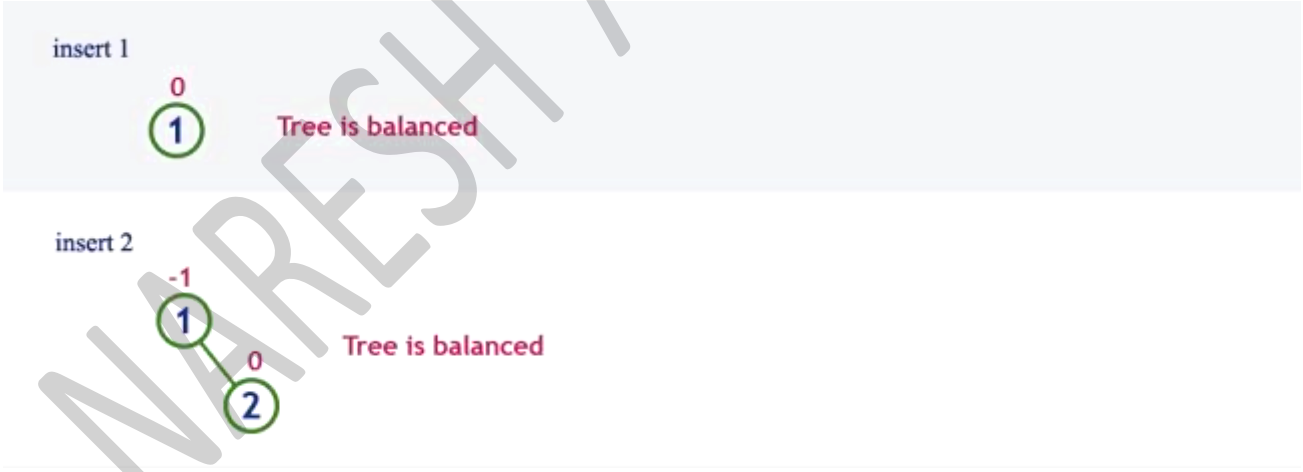
**Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.

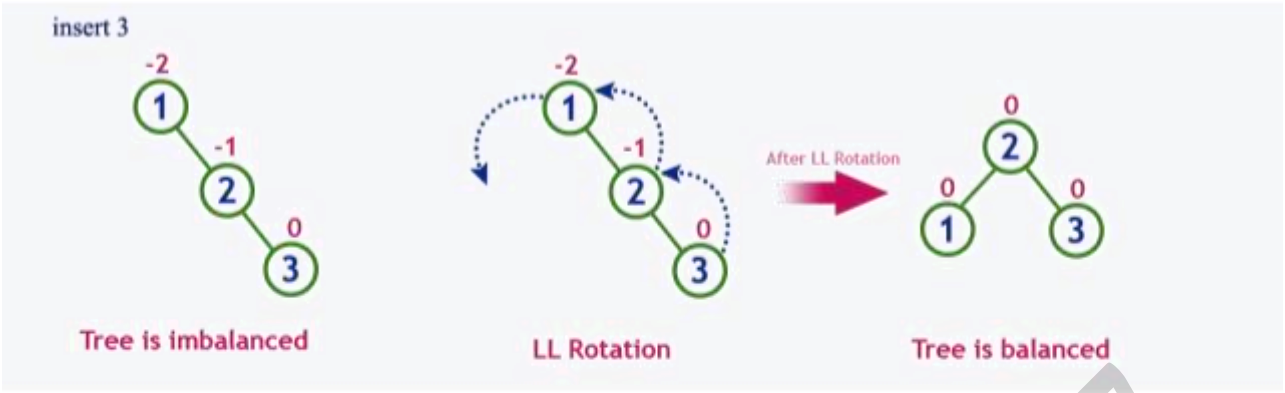
**Step 2 -** After insertion, check the **Balance Factor** of every node.

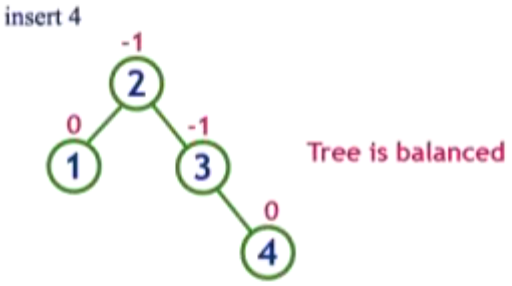
**Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

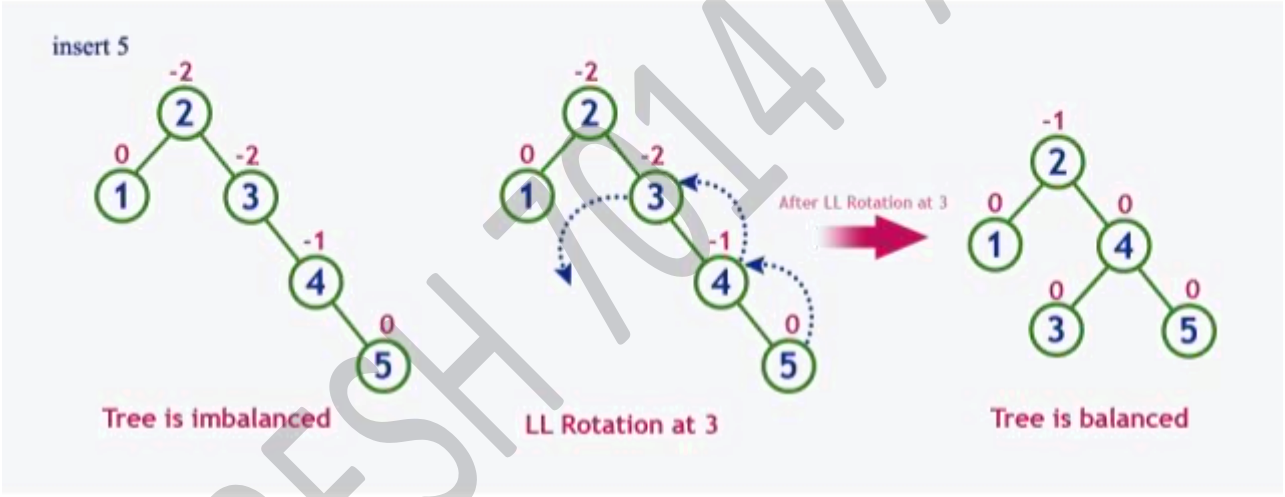
**Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

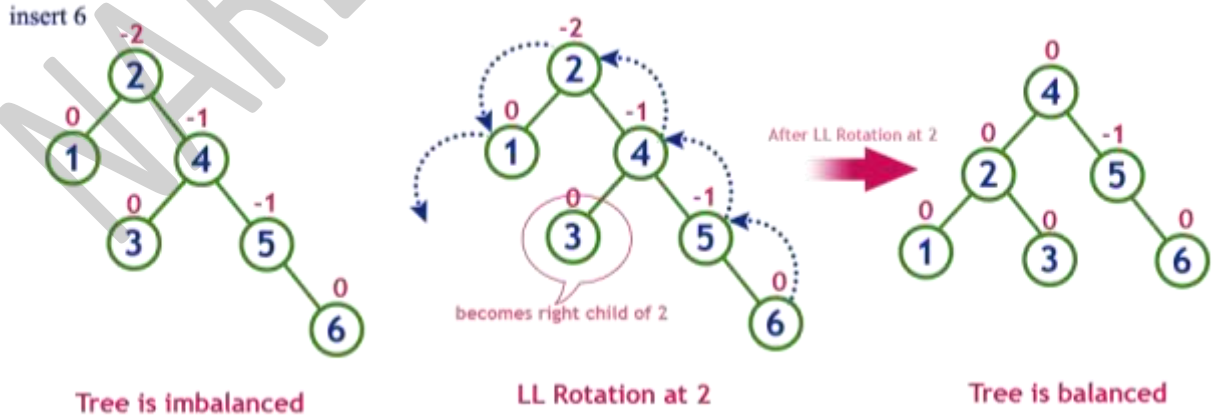
**Example: Construct an AVL Tree by inserting numbers from 1 to 8.**

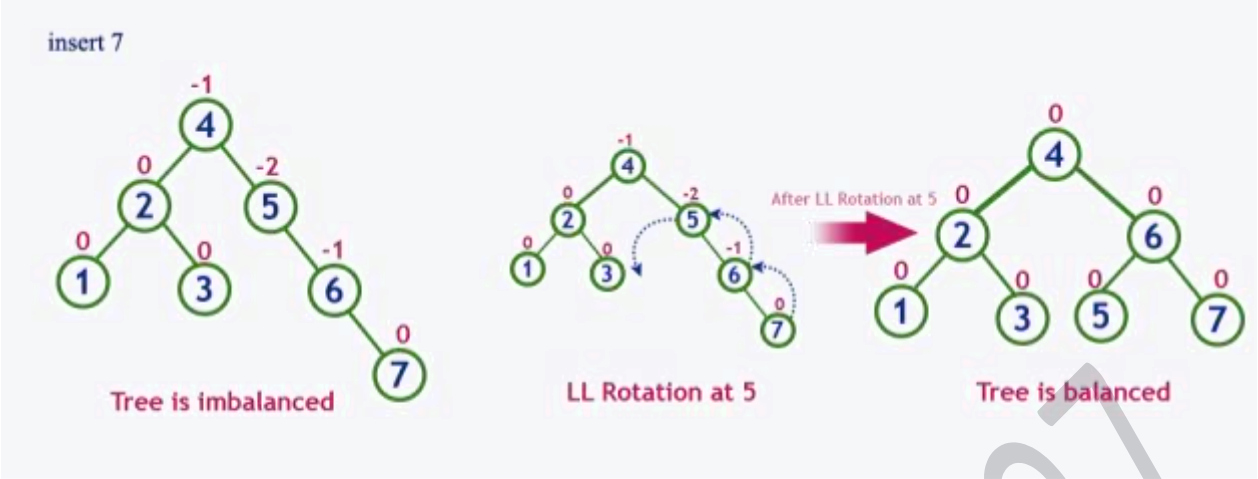


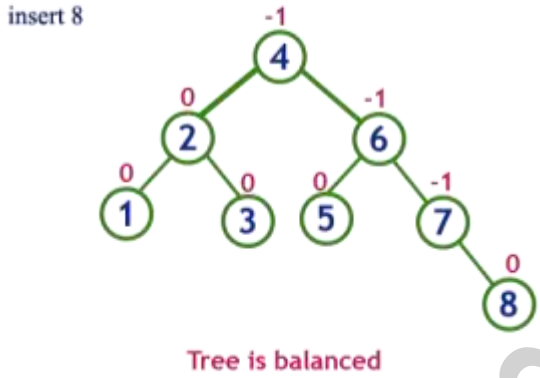





## Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after 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.

# B - Tree

In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.

B-Tree can be defined as follows...

*B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.*

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

**B-Tree of Order m** has the following properties...

**Property #1** - All **leaf nodes** must be **at same level**.

**Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.

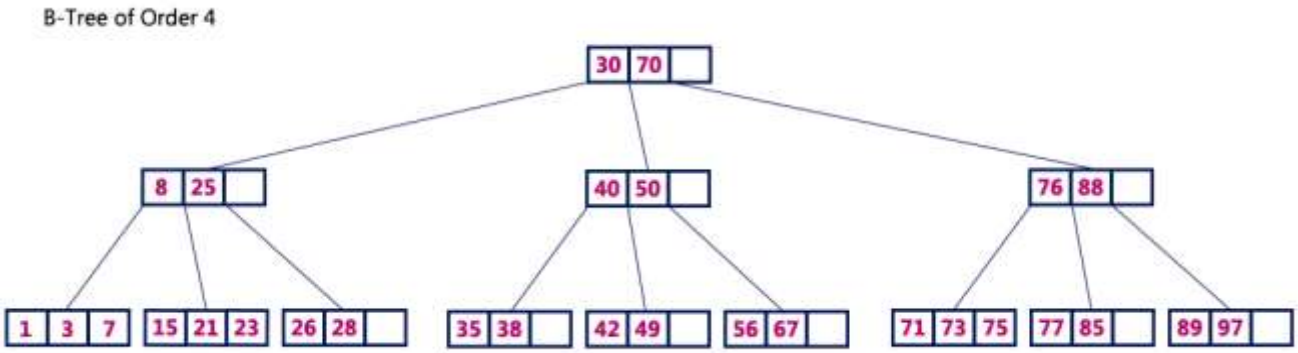
**Property #3** - All non-leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.

**Property #4** - If the root node is a non-leaf node, then it must have **atleast 2** children.

**Property #5** - A non-leaf node with **n-1** keys must have **n** number of children.

**Property #6** - All the **key values in a node** must be in **Ascending Order**.

**For example**, B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.




## Operations on a B-Tree

The following operations are performed on a B-Tree...

1) Search
2) Insertion
3) Deletion

## 1) Search Operation in B-Tree

The search operation in B-Tree is similar to the search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has. In a B-Tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

**Step 1 -** Read the search element from the user.

**Step 2 -** Compare the search element with first key value of root node in the tree.

**Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function

**Step 4 -** If both are not matched, then check whether search element is smaller or larger than that key value.

**Step 5 -** If search element is smaller, then continue the search process in left subtree.

**Step 6 -** If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

**Step 7 -** If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

## 2) Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new **keyValue** is always attached to the leaf node only. The insertion operation is performed as follows...

**Step 1 -** Check whether tree is Empty.

**Step 2 -** If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

**Step 3 -** If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

**Step 4 -** If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

**Step 5 -** If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

**Step 6 -** If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

**Example:** Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

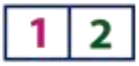
Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

**insert(1)**

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

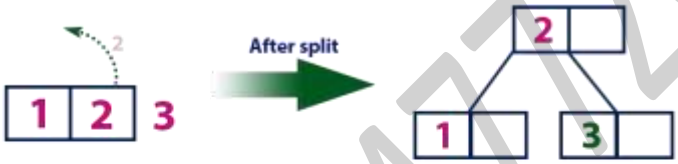

**insert(2)**

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



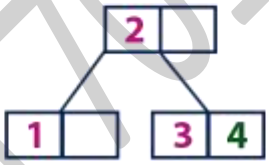

**insert(3)**

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.




**insert(4)**

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

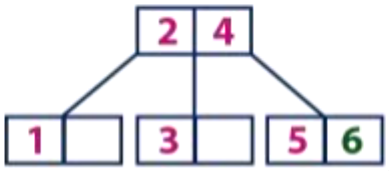


**insert(5)**

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

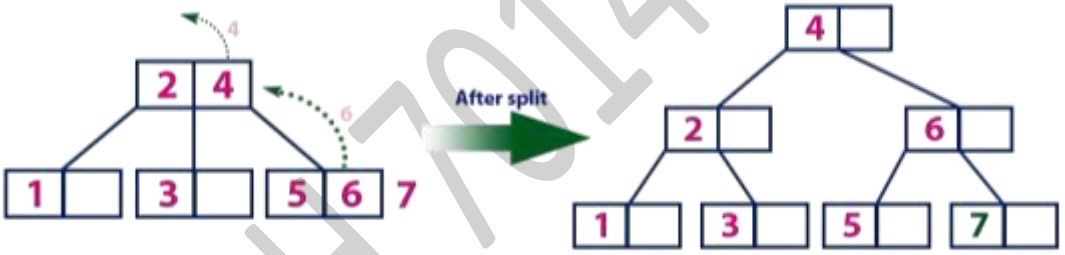

**insert(6)**

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



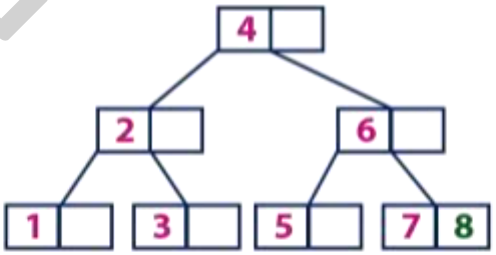

**insert(7)**

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



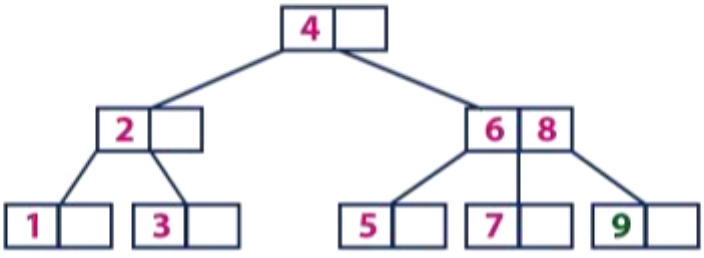

**insert(8)**

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



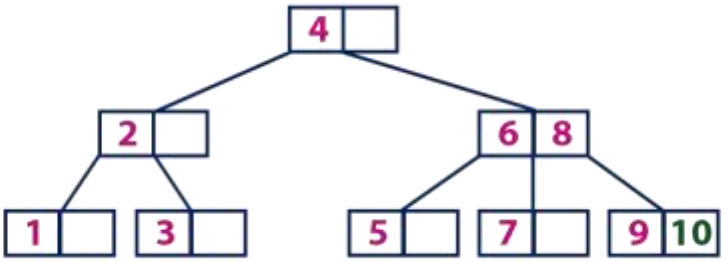

**insert(9)**

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



**insert(10)**

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



# Graphs

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 follows...

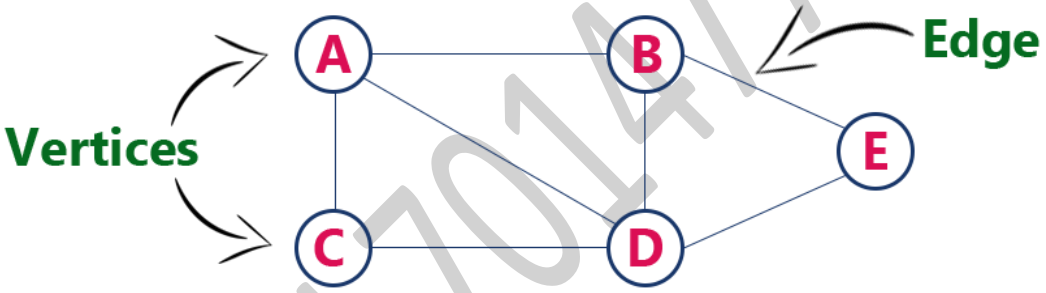
*Graph is a collection of vertices and arcs in which vertices are connected with arcs*

*Graph is a collection of nodes and edges in which nodes are connected with edges*

Generally, a graph **G** is represented as **G = (V, E)**, where **V is set of vertices** and **E is set of edges**.

**Example:** The following is a graph with 5 vertices and 6 edges. This graph G can be defined as G = (V, E)

Where V = {A, B, C, D, E} and E = {(A, B), (A, C) (A, D), (B, D), (C, D), (B, E), (E, D)}.




## Graph Terminology

We use the following terms in graph data structure...

### Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

### Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (starting Vertex, ending Vertex). **For example,** in above graph the link between vertices A and B is represented as (A, B). In above example graph, there are 7 edges (i.e., (A, B), (A, C), (A, D), (B, D), (B, E), (C, D), (D, E)).

### Types of Edges

1) **Undirected Edge -** An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).

2) **Directed Edge -** A directed edge is a unidirectional edge. If there is directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).
3) **Weighted Edge -** A weighted edge is an edge with value (cost) on it.

## Undirected Graph

A graph with only undirected edges is said to be undirected graph.

## Directed Graph

A graph with only directed edges is said to be directed graph.

## Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.

## End vertices or Endpoints

The two vertices joined by edge are called end vertices (or endpoints) of that edge.

## Origin

If an edge is directed, its first endpoint is said to be the origin of it.

## Destination

If an edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

## 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.

## Incident

Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

## Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

## Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

## Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

## 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 outdegree of that vertex.

## Parallel edges or Multiple edges

If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

## Self-loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

## Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

## 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.

# Graph Representations

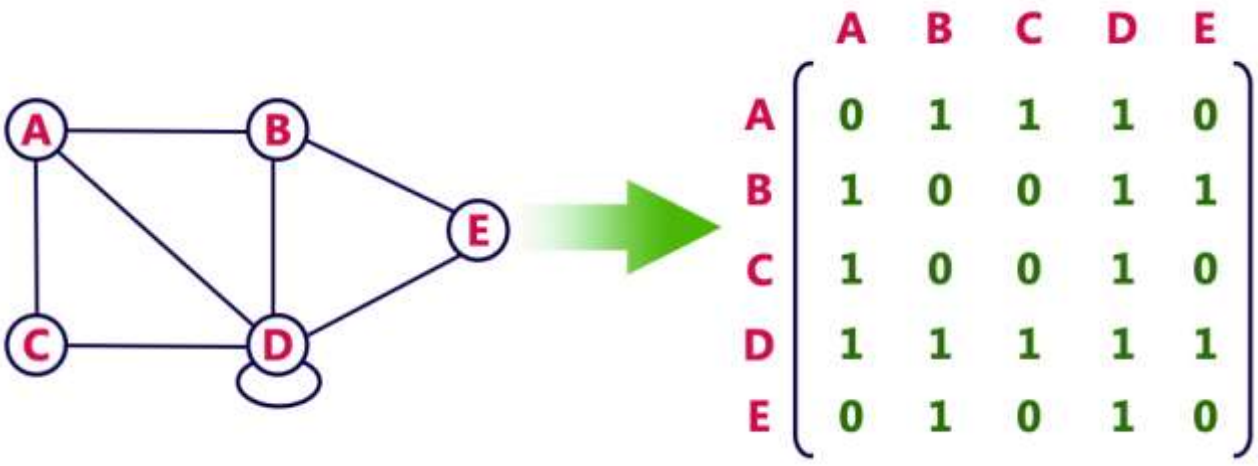
Graph data structure is represented using following representations...

1) Adjacency Matrix
2) Incidence Matrix
3) Adjacency List

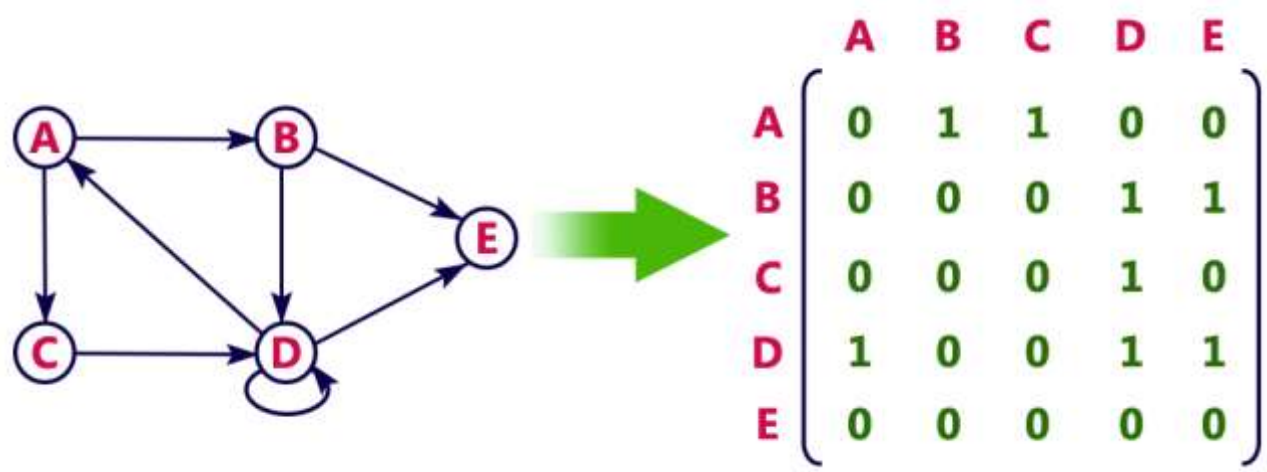
## 1) Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

**For example:** consider the following undirected graph representation...



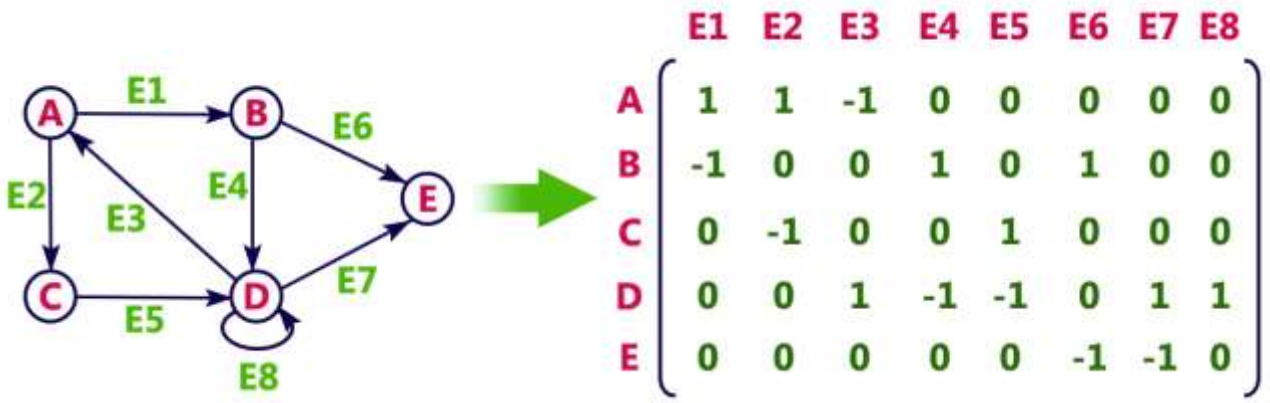

Directed graph representation...




## 2) Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

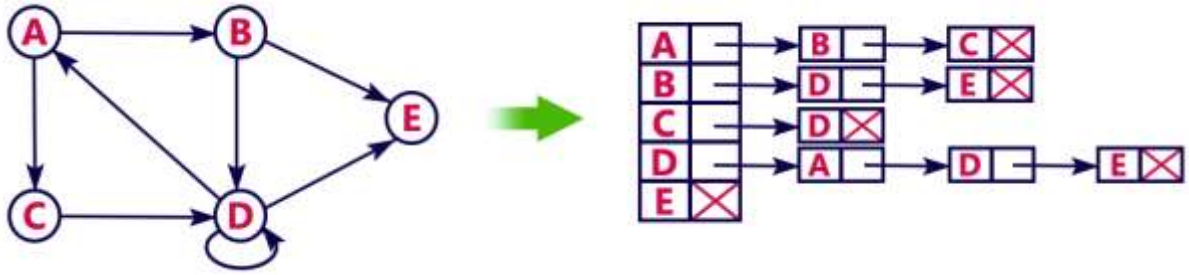
**For example:** consider the following directed graph representation...



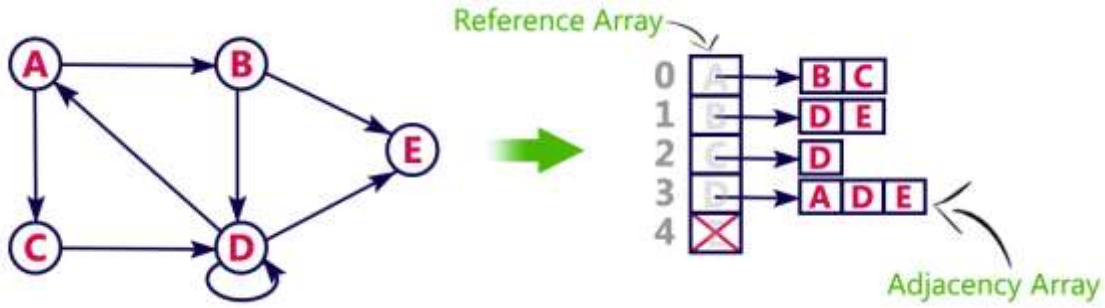

## 3) Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

**For example:** consider the following directed graph representation implemented using linked list...




This representation can also be implemented using an array as given following:




## Graph Traversal - DFS

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also 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. There are two graph traversal techniques and they are as follows...

1) DFS (Depth First Search)
2) BFS (Breadth First Search)

### 1) DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

**Step 1 -** Define a Stack of size total number of vertices in the graph.

**Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

**Step 3 -** Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

**Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

**Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

**Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.

**Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

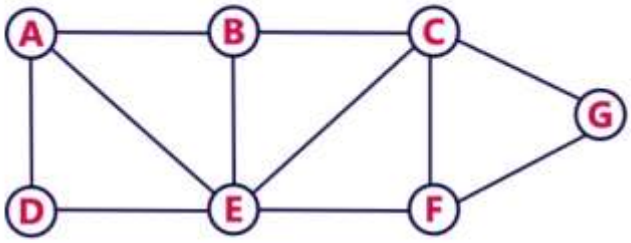
---

**Back tracking** is coming back to the vertex from which we reached the current vertex.

---

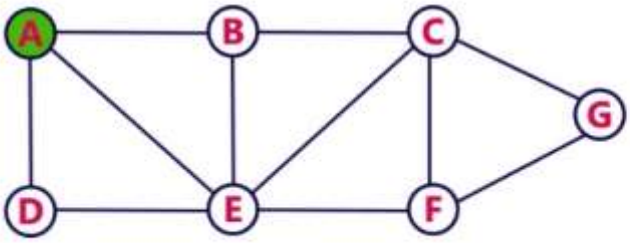
**Example:**

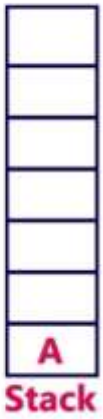
Consider the following example graph to perform DFS traversal




**Step 1:**

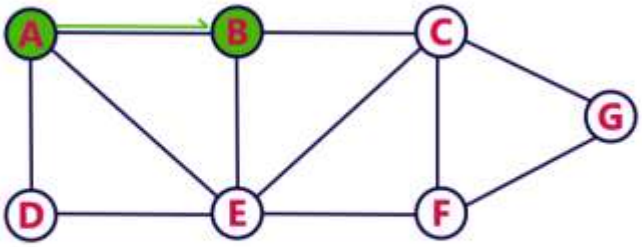
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



**Step 2:**

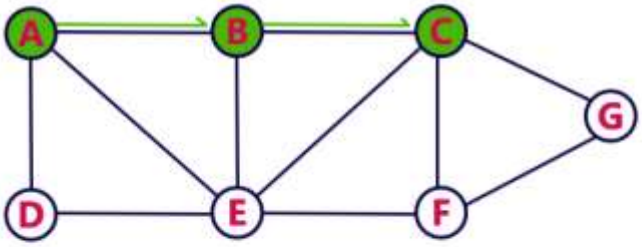
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

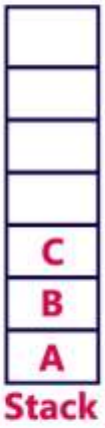




**Step 3:**

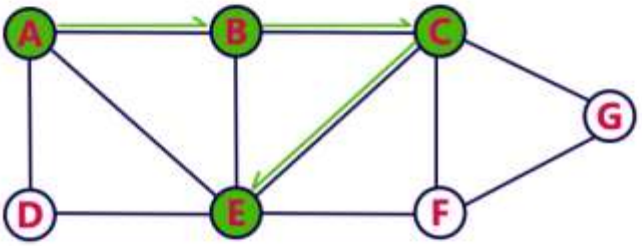
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.




**Step 4:**

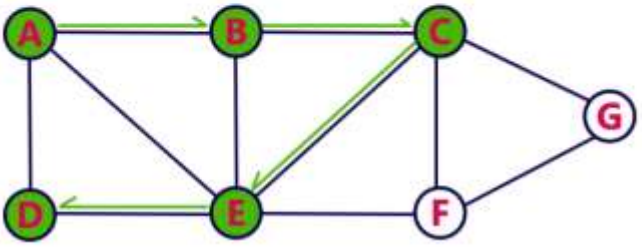
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack





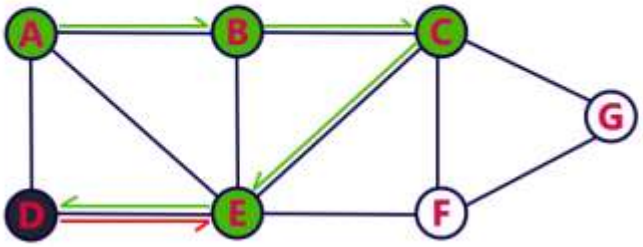
**Step 5:**

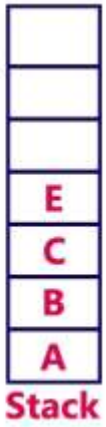
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack




**Step 6:**

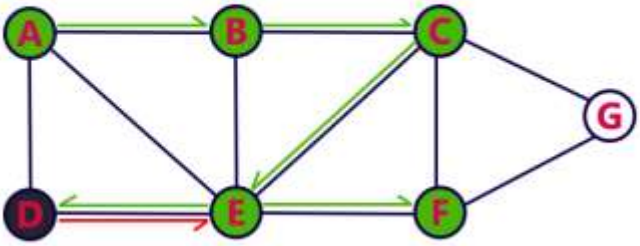
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

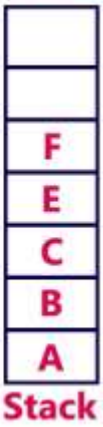



**Step 7:**

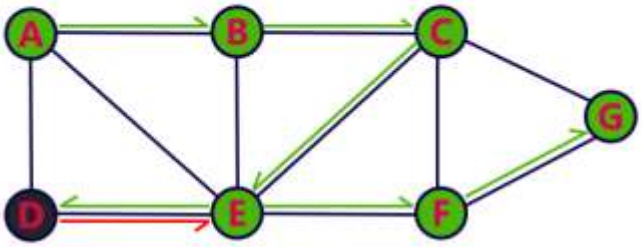
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.




**Step 8:**

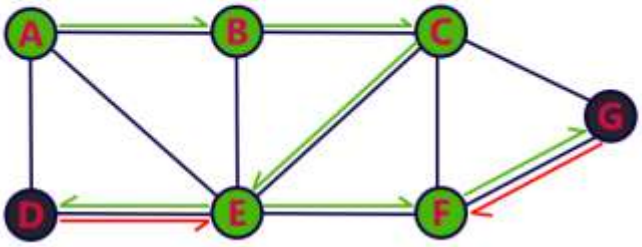
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.





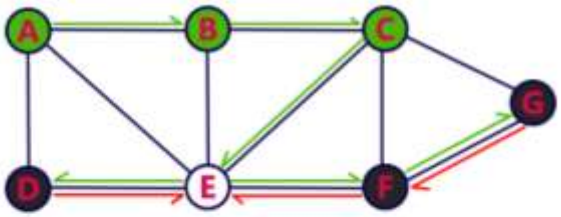
**Step 9:**

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.




**Step 10:**

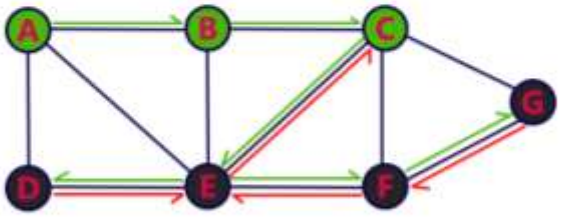
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

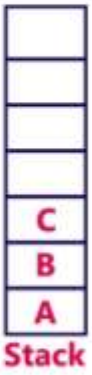




**Step 11:**

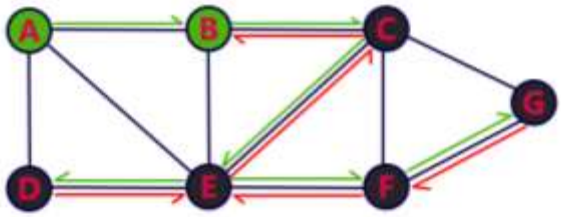
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

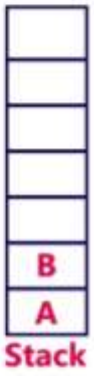



**Step 12:**

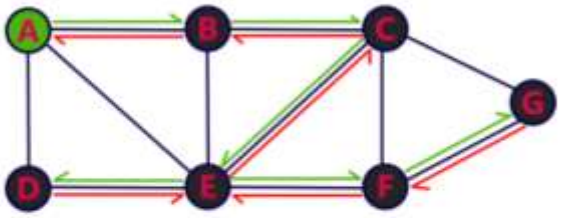
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

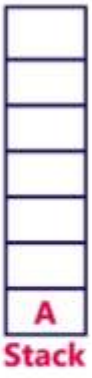



**Step 13:**

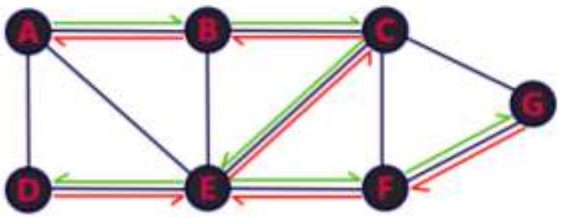
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.




**Step 14:**

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.

## 2) BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal. We use the following steps to implement BFS traversal...

**Step 1 -** Define a Queue of size total number of vertices in the graph.

**Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

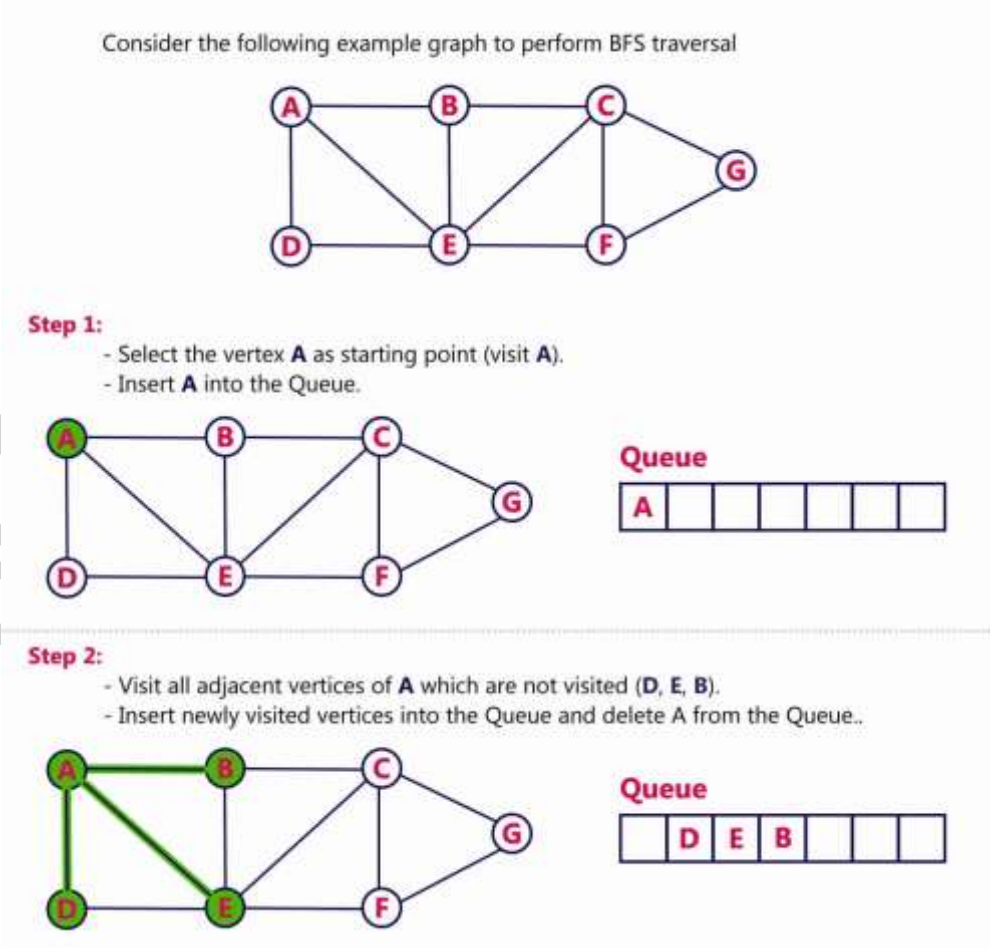
**Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

**Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

**Step 5 -** Repeat steps 3 and 4 until queue becomes empty.

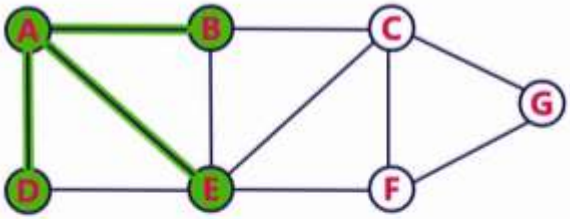
**Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

**Example**



**Step 3:**

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

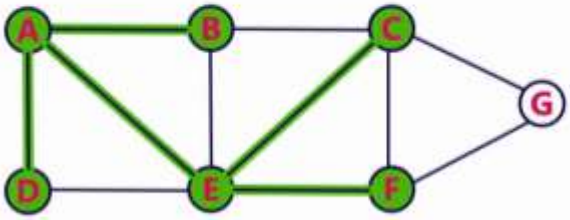



**Queue**

| | | | | | | |
|---|---|---|---|---|---|---|
| | | E | B | | | |

**Step 4:**

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

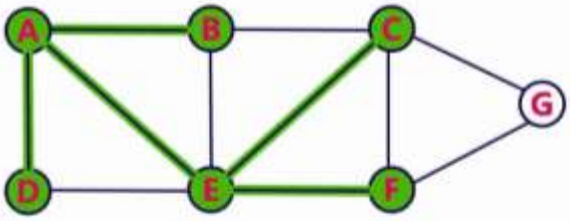



**Queue**

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | B | C | F | |

**Step 5:**

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

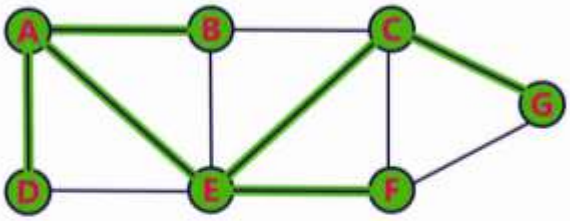



**Queue**

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | C | F | |

**Step 6:**

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

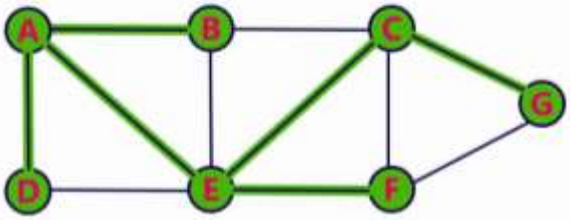



**Queue**

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | F | G |

**Step 7:**

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

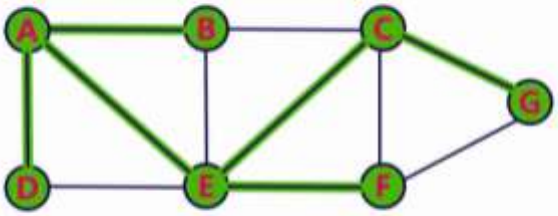



**Queue**

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | G |

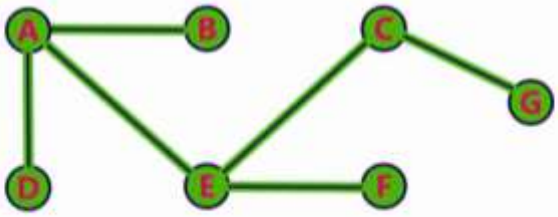
**Step 8:**

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.




**Queue**

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



# Sorting

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

**Telephone Directory −** The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

**Dictionary −** The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

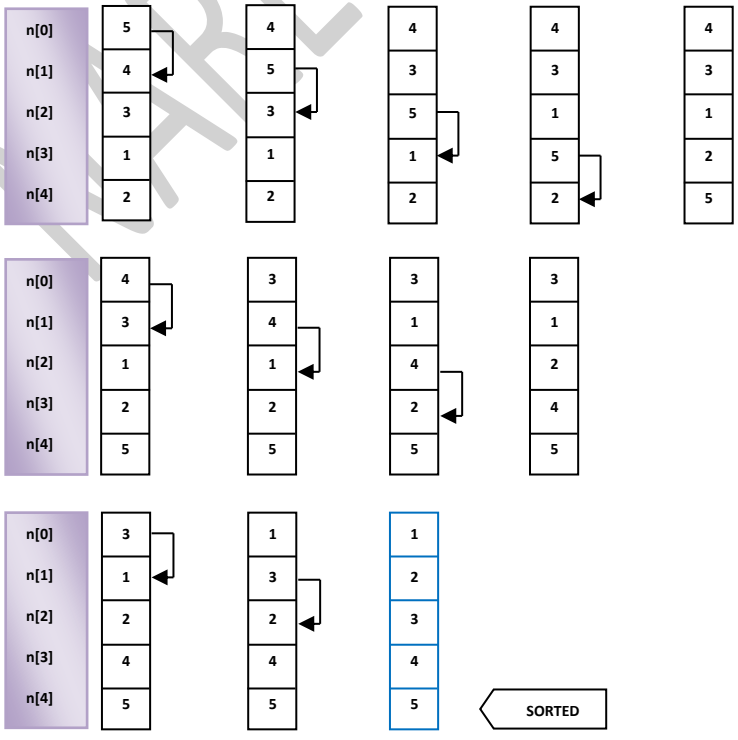
There are various types of sorting available in programming languages, following we are discussing some of them –

1) Bubble Sort
2) Selection Sort
3) Insertion Sort
4) Quick Sort
5) Merge Sort

## 1) Bubble Sort

In this type of sorting technique first the highest number of the array will be sort. In bubble sort, we take element of base address and then respectively check with next address's element if next element is smaller than they make interchange otherwise not. By doing this thing we get our array in sorted order.

**Let see:**



**Que 86: WAP to input 5 element array and sort using Bubble sort.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int n[5],i,j,temp;
   clrscr();
   printf("Enter five elements: ");
   for(i=0;i<=4;i++)
   {
         scanf("%d",&n[i]);
   }
   for(i=0;i<=4;i++)
   {
         for(j=0;j<=3-i;j++)
         {
               if(n[j]>n[j+1])
               {
                     temp=n[j];
                     n[j]=n[j+1];
                     n[j+1]=temp;
               }
         }
   }
   printf("After sorting: \n");
   for(i=0;i<=4;i++)
   {
         printf("%d\t",n[i]);
   }
   getch();
}
```

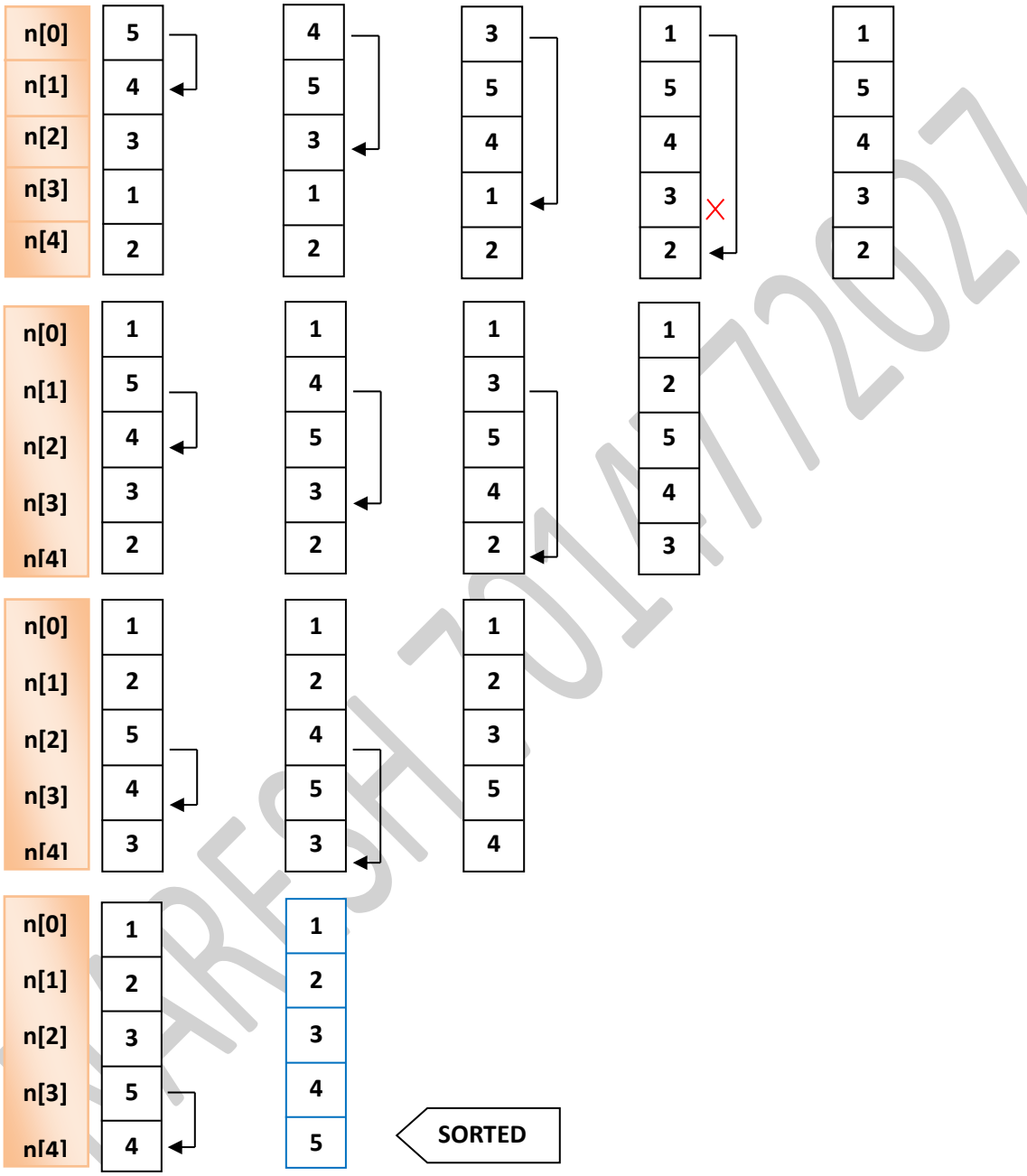
**OUTPUT:**

```
Enter 5 elements: 5
4
3
8
7
After sorting:
3       4       5       7       8
```

## 2) Selection sort

In this sorting, we compare first element with all other element, once base address has smallest element, we start sort for second element and then so on.

**Let see following technique –**



**Que 85: WAP to input 5 element array and sort it using selection sort.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n[5],i,j,temp;
    clrscr();
    printf("Enter five elements: ");
    for(i=0;i<=4;i++)
    {
          scanf("%d",&n[i]);
    }
    for(i=0;i<=4;i++)
    {
          for(j=i+1;j<=4;j++)
          {
                if(n[i]>n[j])
                {
                      temp=n[i];
                      n[i]=n[j];
                      n[j]=temp;
                }
          }
    }
    printf("After sorting: \n");
    for(i=0;i<=4;i++)
    {
          printf("%d\t",n[i]);
    }
    getch();
}
```

**OUTPUT:**

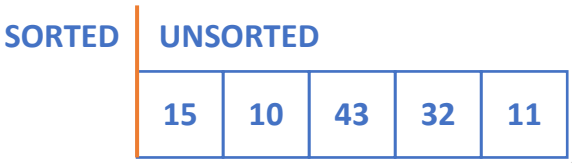
```
Enter 5 elements: 5
4
3
8
7
After sorting:
3     4     5     7     8
```

## 3) Insertion Sort

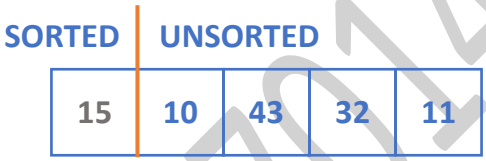
Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list. **Consider the following example-**

Unsorted Array:


Assume that sorted portion of the array is empty and all elements in the array are in unsorted portion of the array as shown in following list.

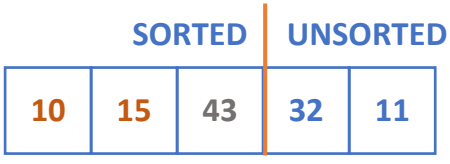



Move the first **element 15** from the unsorted portion to sorted portion of the list.

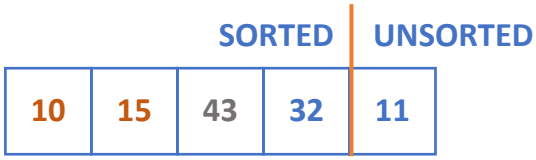



Move the **element 10** from the unsorted portion to sorted portion of the list and compare 10 with 15 and swap them and insert it at right position.


Move the **element 43** from the unsorted portion to sorted portion and compare 43 with 10, 15 and it is larger than both sorted elements so insert it at last position in sorted portion.



Move the **element 32** from the unsorted portion to sorted portion of the list and compare 32 with 43, 10, 15 and since 32 is larger than 15, so move 43 to one position right in the list and insert it after 15 in the sorted portion.




Move the **element 11** from the unsorted portion to sorted portion and compare 11 with 10, 15, 32, 43 and since 11 is larger than 10, so move 15, 32, 43 to one position right in the list and insert 11 after 10 in the sorted portion.


```c
#include <stdio.h>
void main()
{

    int 5, i, j, temp, a[5];
    printf("Enter 5 integer values: ");
    for (i = 0; i < 5; i++)
    {
        scanf("%d", &a[i]);
    }
    //Insertion sort logic
    for (i = 1; i < 5; i++)
    {
        temp = a[i];
        j = i - 1;
        while ((temp < a[j]) && (j >= 0))
        {
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = temp;
    }
    printf("Array after Sorting is: ");
    for (i = 0; i < 5; i++)
    {
        printf(" %d\t", a[i]);
    }
    getch();
}
```