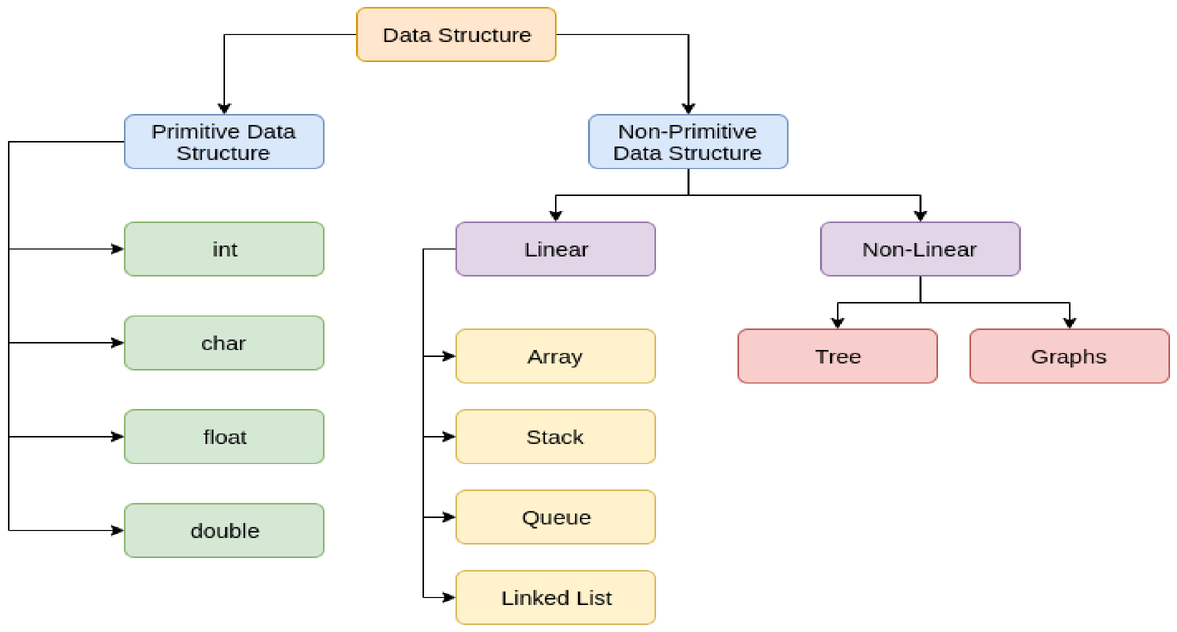
INTRODUCTION TO DATA STRUCTURES

Data Structure:

It is a collection of data items organized in a structure manner, so that data elements can be operated upon efficiently by a set of operations.

TYPES OF DATA STRUCTURE

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 normally classified into two categories.

1. Primitive Data Structure
2. Non-primitive data Structure

1. Primitive Data Structure: Primitive data structures are basic structures and are directly operated upon by machine instructions. Primitive data structures have different representations on different computers. Thesedata types are available in most programming languages as built in type.

- Integer: It is a data type which allows all values without fraction part. We can use itfor whole numbers.
- Float: It is a data type which use for storing fractional numbers.
- Character: It is a data type which is used for character values.
- Pointer: A variable that holds memory address of another variable are called pointer.

2. Non Primitive Data Structure: These are more sophisticated data structures. These are derived from primitive data structures. The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items. A Non-primitive data type is further divided into Linear and Non-Linear data structure.

a. Linear Data Structure: If a data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure.

- **Array**: An array is defined as the collection of similar type of data items stored at contiguous memory locations.
- **Stack:** Stack is a Linear Data structure in which, insertion and deletion operations are performed at one end only. Stack is also called as Last in First out (LIFO) data structure. The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
- **Queue**: The data structure which allow the insertion at one end and Deletion at another end, known as Queue. End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end. Queue is also called as First in First out (FIFO) data structure.
- **Linked list:** Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location the elements are linked using pointers.

b. Non-Linear Data Structure: Nonlinear data structures are those data structure in which data items are not arranged in a sequence. Examples of Non-linear Data Structure are Tree and Graph.

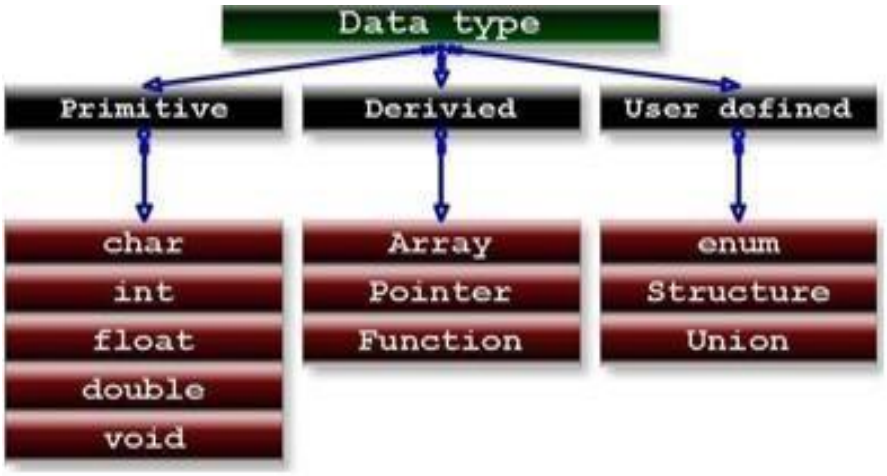
- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.
- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

DIFFERENCE BETWEEN PRIMITIVE AND NON-PRIMITIVE DATA STRUCTURES

PRIMITIVE DATA STRUCTURE	**NON-PRIMITIVE DATA STRUCTURE**
Primitive data structure is a kind of data structure that stores the data of only one type.	Non-primitive data structure is a type of data structure that can store the data of more than one type.
Examples of primitive data structureare integer, character, and float.	Examples of non-primitive data structureare Array, Linked list, stack.
Primitive data structure will containsome value, i.e., it	Non-primitive data structure can consist ofa NULL value.

cannot be NULL.	
The size depends on the type of thedata structure.	In case of non-primitive data structure, size isnot fixed.
It starts with a lowercase character.	It starts with an uppercase character.
Primitive data structure can be used tocall the methods.	Non-primitive data structure cannot be usedto call the methods.

DATA TYPES IN C: What kind of value that the variable will hold is called Data type. 'C' supports several data types of data each of which is stored differently in the computer's memory mainly data types are divided into three types.



1. Primitive Data type: 'C' supports mainly four primitive datatypes.

a) Character Data Type
b) Integer Data Type
c) Float Data Type
d) Double Data Type
e) Void Data Type

a) Character Data Type: The character data type accepts single character only. Characters are either signed or unsigned. But mostly characters are used an unsigned type. The size of the character data type is1 byte in the memory. The range of unsigned character is 0to 255. The range of signed are character is – 128 to + 127. Char is the keyword of the character data type.

Syntax: char list of variables;

Ex: char ch1, ch2, ch3;

b) Integer Data Type: An integer type accepts integer values only. It does not contains any real or float values. The range of an integer variable is -32, 768 to +327,67. int is the keyword for integer data type. In generally 2 bytes of memory is required to store an integer value.

Syntax: int list of variables;

Ex: int a, b, c;

C) Float Data Type: The float data type accepts real values it can contains any floating point values. The range of the floating variable is 3.4E – 38 to 3.4E + 38. Float is the keyword for floating Data type. In generally 4 bytes of memory is required to store an float value with 6 digits of precision.

Syntax: float list of variable;

Ex: float f1, f2, f3;

d) Double Data Type: The double data type accepts large floating values. The range of the double variable is 1.7E –308 to 1.7E + 308. Double is the key word for double data type. In generally 8 bytes of memory is required to store double value.

Syntax: double list of variables;

Ex: double d, e, f;

e) Void Data type: Void is an empty data type that has no value. . The void keyword specifies that the function does not return a value.

2. Derived Data Types: Derived data types are derived from the primary data types. The derived data types may be used for representing a single or multiple values. These are called secondary data type. The derived data types are arrays, pointers, functions, etc.

- **Array:** An array is defined as the collection of similar type of data items stored at contiguous memory locations.
- **Pointer:** A pointer is a variable that stores the address of another variable.
- **Function:** A function is a group of statements that together perform a task. Every Cprogram has at least one function, which is main ().

3. User Defined Data type: The C language allows a feature known as the type definition. This basically allows a programmer to provide a definition to an identifier that will represent a data type which already exists in a program. The C program consists of the following types:

a) Structures
b) Union
c) Typedef
d) enum

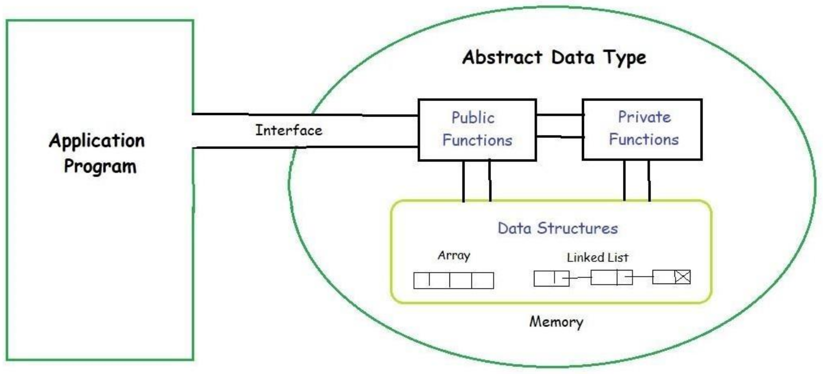
a) Structures: We use the structure for organizing a group of various data items into one single entity – for grouping those data items that are related but might belong to different data types.

b) Union: A union is basically a collection of various different data types that are present in the C language, but not very similar to each other. The union mainly allows the storage of those data types that are different from each other in the very same memory location.

c) Typedef: We use the keyword typedef for creating an alias (a new name) for a data type that already exists. The typedef won't create any new form of data type.

d) enum: The enum refers to a keyword that we use for creating an enumerated data type. The enum is basically a special data type (enumerated data type) that consists of a set of various named values – known as members or elements. We mainly use it for assigning different names to the integral constants. This way, the program becomes much more readable.

ABSTRACT DATA TYPE(ADT): Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.



The user of data type does not need to know how that data type is implemented, for example, we have beenusing Primitive values like int, float, and char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs to know what a data type can do, but not how it will be implemented. This is known as ADT.

Operations on ADT:

- **Find (key):** Return a record with the given key or Null if no record with the given key.
- **Insert (key, data):** Insert a new record with the given key and error if the dictionary already contains a record with the given key.
- **Remove(key):** Removes the record with the given key and error if there is no record with the given key.

ARRAYS

Array: An array is a collection of similar data items stored in continuous memory locations with single name.

TYPES OF ARRAYS

In c programming language, arrays are classified into two types. They are as follows...

1. Single Dimensional Array / One Dimensional Array
2. Two-Dimensional Array

SINGLE DIMENSIONAL ARRAY / ONE DIMENSIONAL ARRAY: In c programming language, single dimensional arrays are used to store list of values of same data type. In other words, single dimensional arrays are used to store a row of values. In single dimensional array, data is stored in linear form. Single dimensional arrays are also called as one-dimensional arrays, Linear Arrays or simply 1-D Arrays.

Declaration of Single Dimensional Array: We use the following general syntax for declaring a single dimensional array.

Syntax: data type array name[size];

Example code: int rollNumbers[60];

The above declaration of single dimensional array reserves 60 continuous memory locations of 2 byteseach with the name roll Numbers and tell the compiler to allow only integer values into those memory locations.

Initialization of Single Dimensional Array: We use the following general syntax for declaring and initializing a single dimensional array with size and initialvalues.

Syntax: Data type array name[size] = {value1, value2, ...} ;

Example code: int marks [6] = { 89, 90, 76, 78, 98, 86 } ;

The above declaration of single dimensional array reserves 6 contiguous memory locations of 2 bytes each with the name marks and initializes with value 89 in first memory location, 90 in second memory location, 76 in third memory location, 78 in fourth memory location, 98 in fifth memory location and 86 in sixth memory location.

Accessing Elements of Single Dimensional Array

In c programming language, to access the elements of single dimensional array we use array name followed by index value of the element that to be accessed. Here the index value must be enclosed in squarebraces. Index value of an element in an array is the reference number given to each element at the time of

memory allocation. The index value of single dimensional array starts with zero (0) for first element and incremented by one for each element. The index value in an array is also called as subscript or indices. We use the following general syntax to access individual elements of single dimensional array...

Syntax: arrayName[Index value];

Example code: marks [2] = 99;

In the above statement, the third element of 'marks' array is assigned with value '99'.

TWO DIMENSIONAL ARRAY: The Two dimensional arrays are used to store data in the form of table. We also use 2-D arrays to createmathematical **matrices**.

Declaration of Two Dimensional Array: We use the following general syntax for declaring a two dimensional array...

Syntax: Data type array name[row Size] [column Size] ;

Example Code: int matrixA [2][3] ;

The above declaration of two dimensional array reserves 6 continuous memory locations of 2 bytes eachin the form of **2 rows** and **3 columns**.

Initialization of Two Dimensional Array: We use the following general syntax for declaring and initializing a two dimensional array with specific numberof rows and coloumns with initial values.

Syntax: Data type array name [rows][colmns] = {{r1c1value, r1c2value, ...},{r2c1, r2c2,...}...} ;

Example Code: int matrix A [2][3] = { {1, 2, 3},{4, 5, 6} } ;

The above declaration of two-dimensional array reserves 6 contiguous memory locations of 2 bytes each in the form of 2 rows and 3 columns. And the first row is initialized with values 1, 2 & 3 and second row is initialized with values 4, 5 & 6. We can also initialize as follows...

Example Code

```
int matrix_A [2][3] = {
                 {1, 2, 3},
                 {4, 5, 6}
                } ;
```

Accessing Individual Elements of Two Dimensional Array

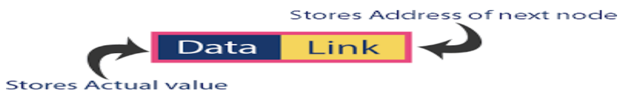
In a c programming language, to access elements of a two-dimensional array we use array name followed by row index value and column index value of the element that to be accessed. Herethe row and column index values must be enclosed in separate square braces. In case of the two- dimensional array the compiler assigns separate index values for rows and columns. We use the following general syntax to access the individual elements of a two-dimensional array.

Syntax: arrayName[rowIndex] [columnIndex]

Example code: matrixA[0][1] = 10 ;

In the above statement, the element with row index 0 and column index 1 of **matrixA** array is assigned withvalue **10**.

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".



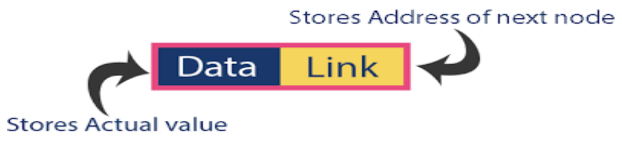
TYPES OF LINKED LIST: There are four key types of linked lists:

1. Single linked list
2. Double linked list
3. Circular linked list
4. Circular double linked list

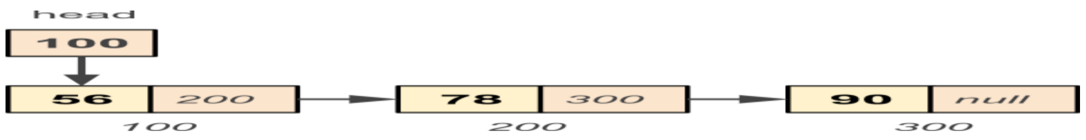
Linked List
Singly
Doubly
Circular
Circular Doubly

1. Single linked list: A singly linked list is a linear data structure in which the elements are stored in sequential format and each element is connected only to its next element using a pointer. Each element in a linked list is called a node. Every Node contains two fields, data field, and **Link** or **Pointer**. The data fieldis used to store actual value of the node and **Link** or **Pointer** 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...



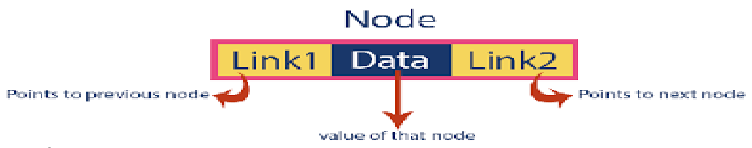
Example:



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...

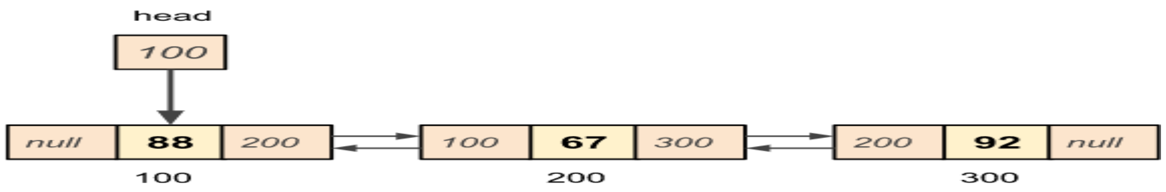
"Double linked list is a sequence of elements in which every element has links to its previous element andnext element in the sequence."

In a double linked list, every node has a link to its previous node and next node. So, we can traverse

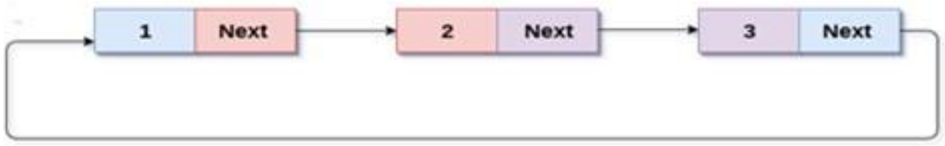


forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...

Example

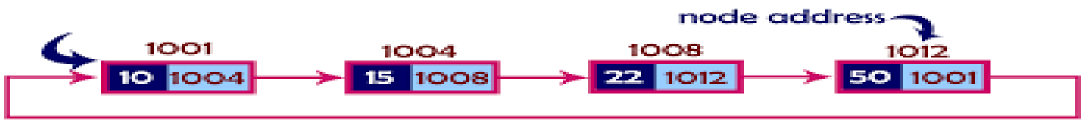


3. Circular linked list: The circular linked list is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.

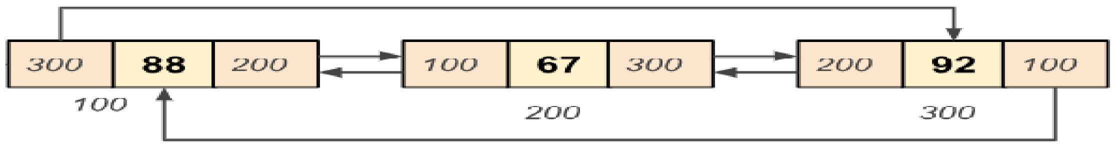


There are generally two types of circular linked lists:

1. Circular singly linked list: In a circular Singly linked list, the last node of the list contains a pointer tothe first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.

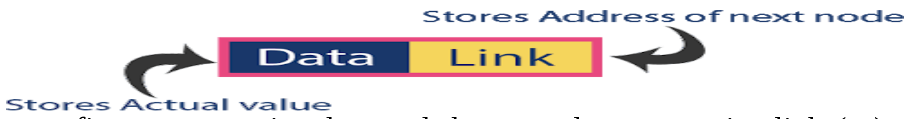


Circular Doubly linked list: Circular doubly linked list is a linear data structure, in which the elements are stored in the form of a node. Each node contains three sub-elements. A data part that stores the value of the element, the previous part that stores the pointer to the previous node, and the next part that stores the pointer to the next node A circular doubly linked list is shown in the following figure.



In Circular double linked list, last element contains link to the first element as next and the first element contains link of the last element as previous. A circular doubly linked can be visualized as a chain of nodes, where every node points to previous and next node.

SINGLE LINKED LIST: A linked list is a linear collection of data items is called as Nodes. Each node



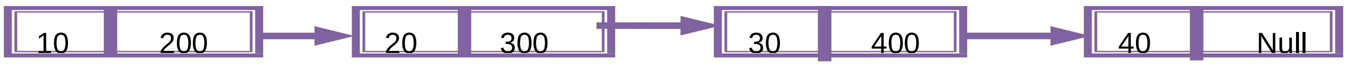
divided into two parts, first part contains data and the second part contains link (or) pointer. This link or pointer contains address of the next node in the list.

Example:

For example, the first part contains data items such as 10,20,30,40 and other part contains address of the next node such as 100,200,300 and 400. The last node contains null that indicates, it is the last node in the list

Example:

Node



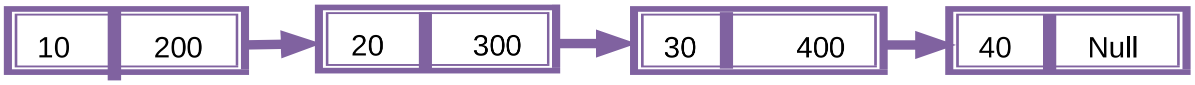
100 **200** **300** **400**

Operations on Linked list: There are 3 operations performed on linked list,

1. Insertion of a node
2. Deletion of a node
3. Traversing a linked list

1. Insertion of a node: Insertion of a node is nothing but adding a new node to the linked list. The addition ofanode done in 3 positions,

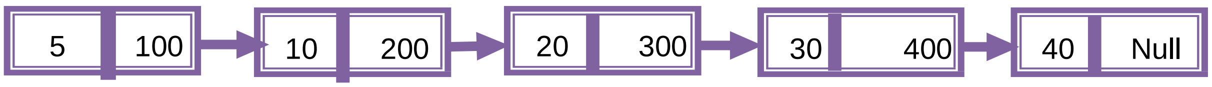
a) At the beginning of the list
b) At the end of the list
c) At the middle of the list



Given list is,

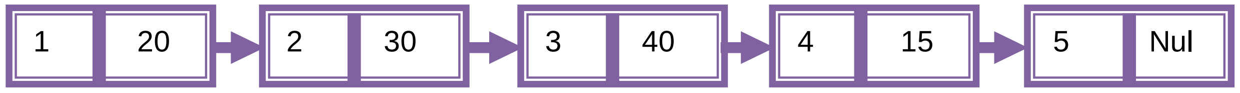
100 **200** **300** **400**

a) At the beginning of the list:



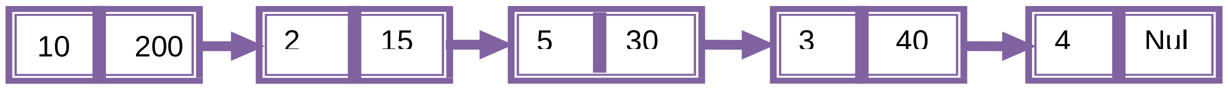
150 100 200 300 400

b) At the end of the list:



100 200 300 400 150

c) At the middle of the list:



100 200 150 300 400

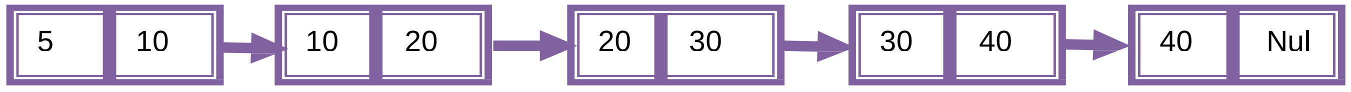
2. Deletion of a Node: Deletion of a node is removing the node from the linked list. Deletion of a nodefrom thelinked list can be done in three positions.

a. At the beginning of the list

b. At the end of the list

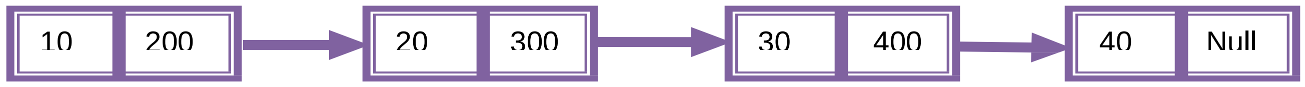
c. At the middle of the

listGiven list is,



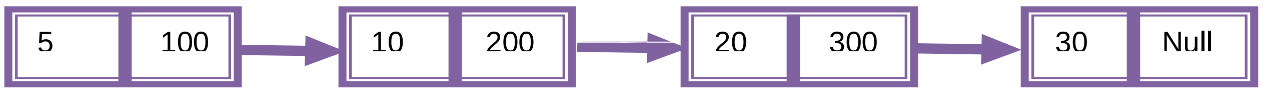
150 100 200 300 400

a. At the beginning of the list:



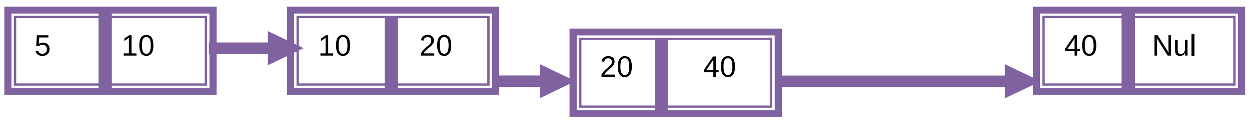
100 200 300 400

b. At the end of the list:



150 100 200 300

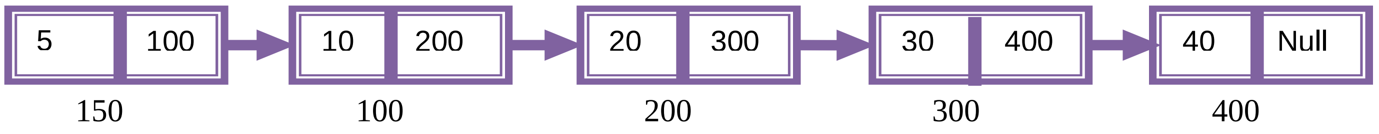
c. At the middle of the list:



150 100 200 400

3. Traversing a linked list: Traversing a linked list means visiting all the nodes of the list for the purpose offinding the particular data, printing data items or to count number of nodes in the list.

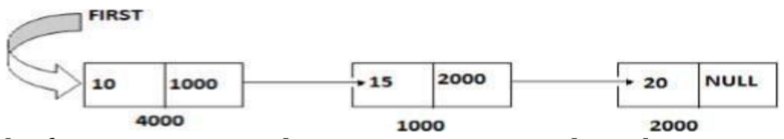
Example:



concept of Single Linked List :

<u>Single Linked List</u> :

In single linked list, the node can be divided into two parts. The first part contains data and second part contains link that which contains address of the next node.



In the above example the first part contains data items 10,15,20 and another part contains address of the next node that is 4000, 1000, 2000.

Operations on Single Linked List : The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

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...

- Inserting at Beginning of the list
- Inserting at End of the list
- Inserting at Specific location in the list

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 new Node with given value.

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

Step 3 - If it is Empty then, set new Node→next = NULL and head = newNode. Step 4 - If it is Not Empty then, set new Node→next = head and head = new Node.

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 new Node with given value and new Node → next as NULL.

Step 2 - Check whether list is Empty (head == NULL). Step 3 - If it is Empty then, set head = new Node.

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 last node in the list (until temp→ next is equal to NULL).

Step 6 - Set temp → next = new Node.

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 new Node with given value.

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

Step 3 - If it is Empty then, set new Node → next = NULL and head = new Node. 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 new Node (until temp1 → data is equal to location, here location is the node value after which we want to insert the new Node).

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 'new Node → next = temp → next' and 'temp → next = new Node'.

Deletion

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

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

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.

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, defines 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.

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, defines 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)).

Displaying a Single Linked List

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).

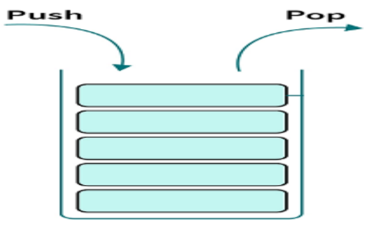
DIFFERENCES BETWEEN ARRAYS AND LINKED LISTS

ARRAY	**LINKED LIST**
1. An array is a collection of elements of a similar data type.	1. Linked List is an ordered collection of elements of the same type in which each element is connected to the next using pointers.
2. Array elements can be accessed randomly using the array index.	2. Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.
3. Data elements are stored in contiguous locations in memory.	3. New elements can be stored anywhere and areference is created for the new element usingpointers.
4. Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	4. Insertion and Deletion operations are fast and easyin a linked list.

5. Memory is allocated during the compile time (Static memory allocation).	5. Memory is allocated during the run-time (Dynamicmemory allocation).
6. Size of the array must be specified at the time of array declaration/initialization.	6. Size of a Linked list grows/shrinks as and whennew elements are inserted/deleted.

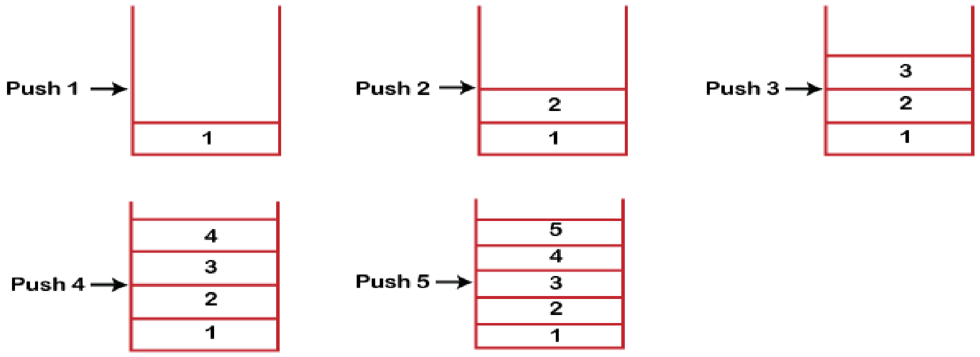
STACK

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.



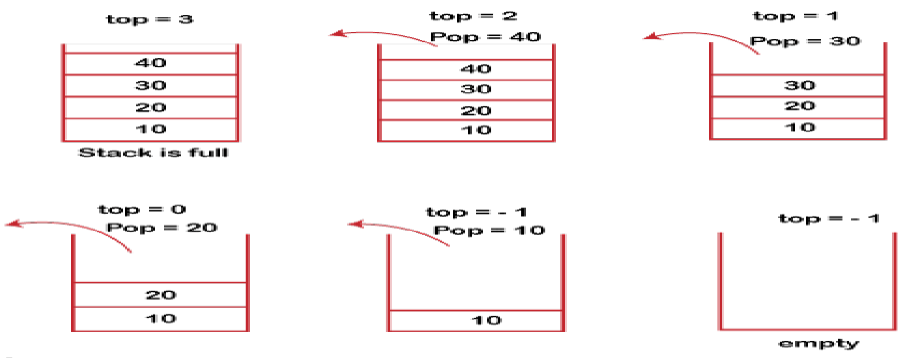
Working of Stack: Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



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.

We have to use the pop operation to delete the elements in the stack. So just mention pop() don't write arguments in the pop because by default it deletes the top element. The first '40' element is deleted next ‘30’ element…..’ 10’. When the top elements are deleting then the top value decreases. When top=-1 the stack indicates underflow. The pop operation is shown in the below figure.



Applications of a Stack

- ❖ Stacks can be used for expression evaluation.
- ❖ Stacks can be used to check parenthesis matching in an expression.
- ❖ Stacks can be used for Conversion from one form of expression to another.
- ❖ Stacks can be used for Memory Management.
- ❖ Stack data structures are used in backtracking problems.
- ❖ Calculating (computing) prefix and postfix notations.

STACK OPERATIONS: The following are some common operations implemented on the stack:

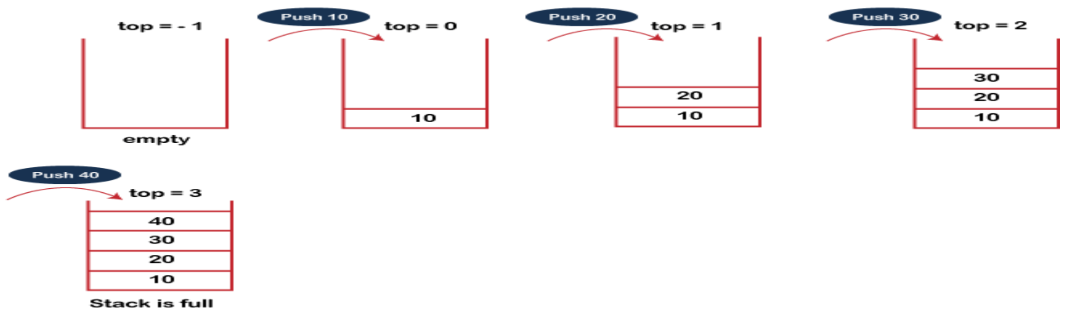
1. push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
2. pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
3. display(): It prints all the elements available in the stack.

push():

1. 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.

Example:

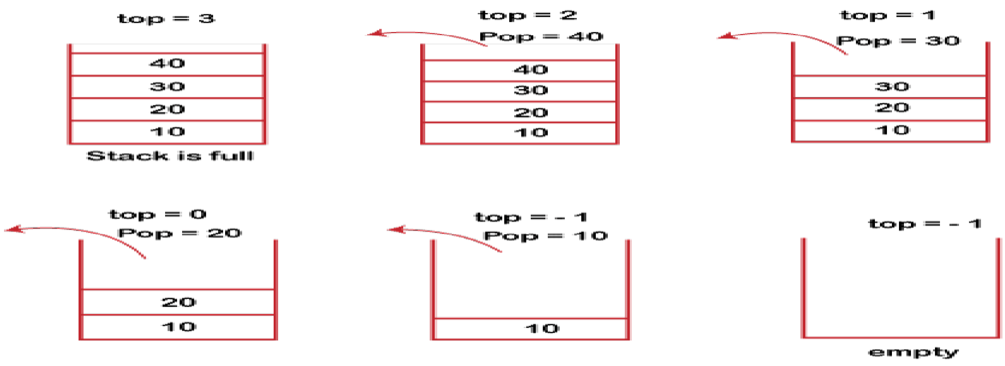
The elements are inserted in the order as 10, 20, 30, 40, it represents the stack of four elements. In figure (A), we want to push ‘10’ element on the stack then the top becomes zero (top=0), similarly the top=1 when ‘20’ element is pushed, top=2 when the ‘30’ element is pushed, top=3 when the ‘40’ element is pushed.



So whatever the elements we have taken is placed in the stack, now the stack is full. If you want to push another element there is no place in the stack, so it indicates the overflow. Now the stack is full if you want to pop the element '40' element has to be deleted first. The push operation is shown in the below figure.

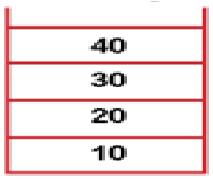
2. pop(): In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position.

We have to use the pop operation to delete the elements in the stack. So just mention pop() and don't write arguments in the pop because by default it deletes the top element. The first '40' element is deleted next '30' element.' 10'. When the top elements are deleting then the top value decreases. When top=-1 the stack indicates underflow. The pop operation is shown in the below figure(B).



3. **display ():**It prints all the elements available in the stack.

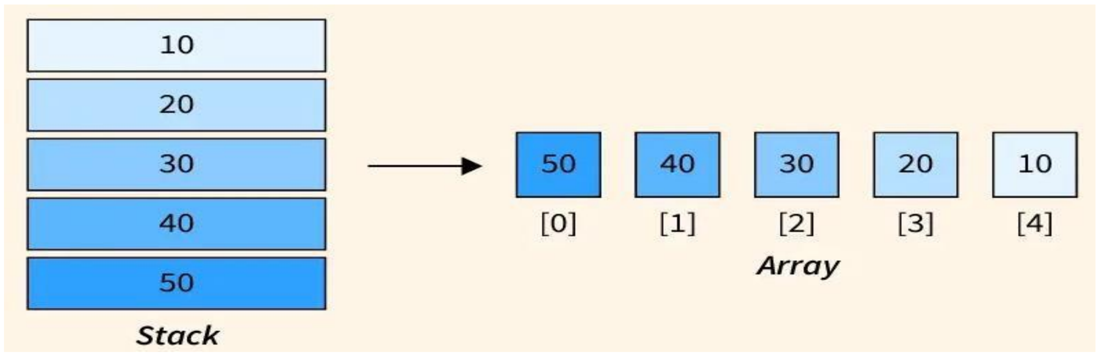
It displays elements as follows **40, 30, 20, 10.**



IMPLEMENTATION OF STACK: There are two ways to implement a stack –

- **Using array**
- Using linked list

IMPLEMENT A STACK USING AN ARRAY: In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Let's see how each operation can be implemented on the stack using array data structure. 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.



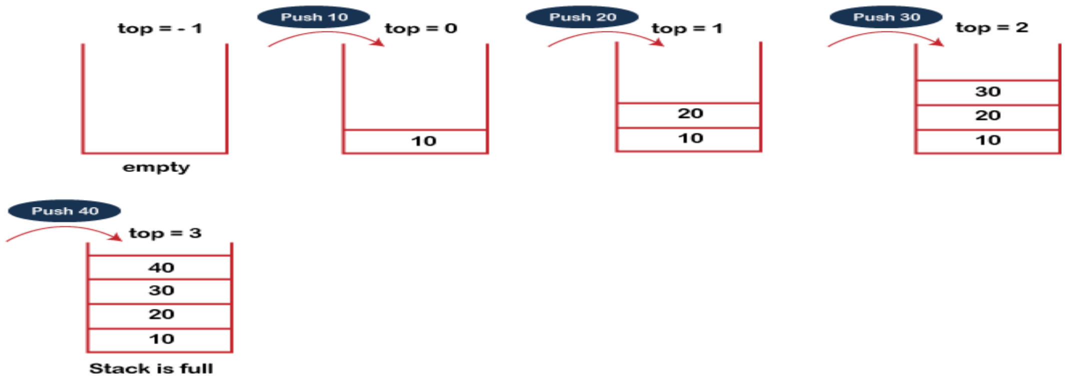
Stack Operations using Array: The following operations are performed on the stack,

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)

1. Push (To insert an element on to 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.

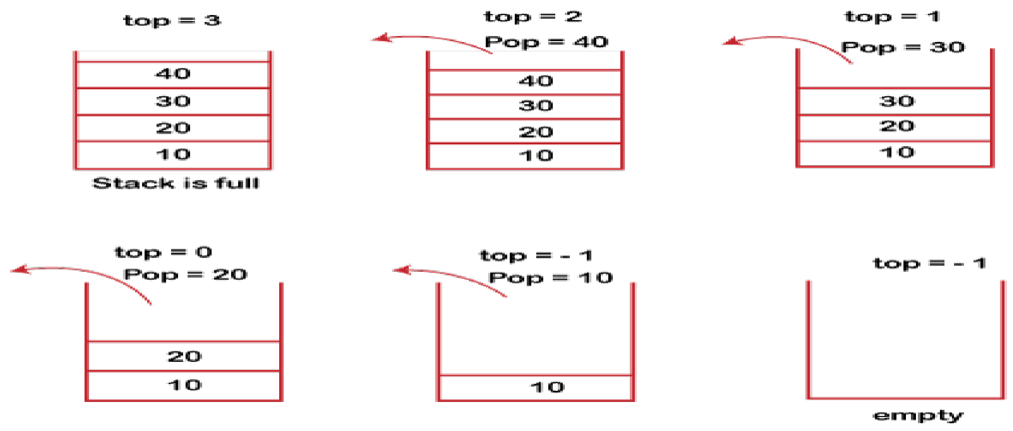
Example:

The elements are inserted in the order as 10, 20, 30, 40, it represents the stack of four elements. In figure (A), we want to push '10' element on the stack then the top becomes zero (top=0), similarly the top=1 when '20' element is pushed, top=2 when the '30' element is pushed, top=3 when the '40' element is pushed. So whatever the elements we have taken is placed in the stack, now the stack is full. If you want to push another element there is no place in the stack, so it indicates the overflow. Now the stack is full if you want to pop the element '40' element has to be deleted first. The push operation is shown in the below figure.



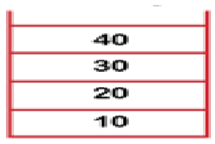
2. Pop (To delete an element from the stack): In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position.

We have to use the pop operation to delete the elements in the stack. So just mention pop() and don't write arguments in the pop because by default it deletes the top element. The first '40' element is deleted next '30' element…..' 10'. When the top elements are deleting then the top value decreases. When top=-1 the stack indicates underflow. The pop operation is shown in the below figure(B).



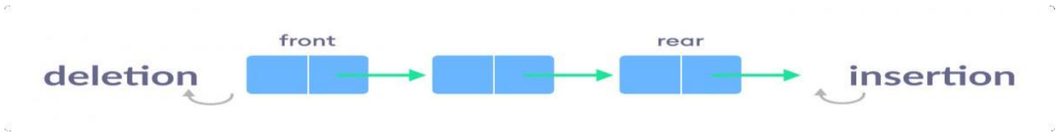
3. Display (To display elements of the stack): It prints all the elements available in the stack.

It displays elements as follows **40, 30, 20, 10.**



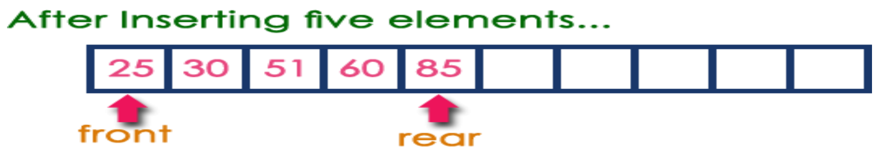
QUEUE

QUEUE: Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. The insertion is performed at one end 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.



Example

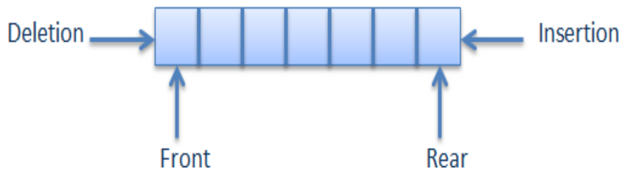
Queue after inserting 25, 30, 51, 60 and 85.



TYPES OF QUEUES: There are five different types of queues that are used in different scenarios. They are:

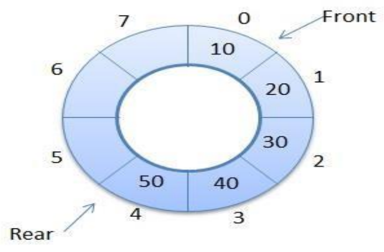
1. Simple Queue
2. Circular Queue
3. Double Ended Queue (Deque)
4. Priority Queue

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

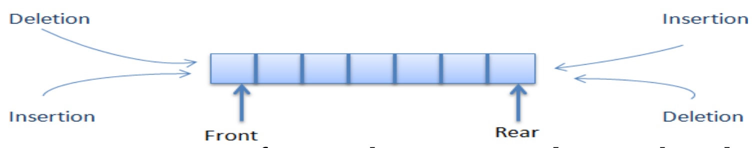


The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

2. Circular queue: 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. It is also called **'Ring Buffer'**.

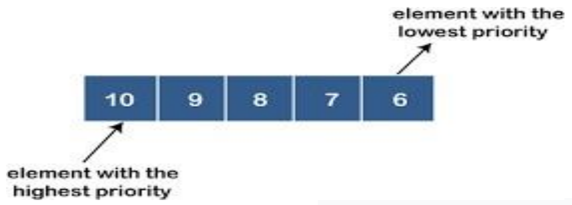


3. Double Ended Queue (Deque): Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions. Since Deque supports both stack and queue operations, it can be used as both.



Priority Queue: A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are retrieved before elements with lower priority values.

In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value. For example, if you add an element with a high priority value to a priority queue, it may be inserted near the front of the queue, while an element with a low priority value may be inserted near the back.


element with the lowest priority
10
9
8
7
6
element with the highest priority

IMPLEMENTATION OF QUEUES USING ARRAY: Implementation of queue using array starts with the creation of an array of size n. And initialize two variables FRONT and REAR with-1which means currently queue is empty.

Queue Operations using Array: Now, some of the implementations of queue operations are as follows:

1. enQueue()
2. deQueue()
3. display():

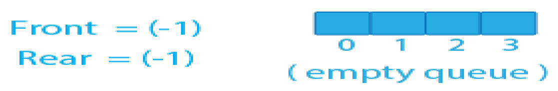
1. enQueue(value) - Inserting value into the queue: In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue.

2. 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.

3. Display: Print all elements of the queue. If the queue is non-empty, traverse and print all the elements from the index front to rear.

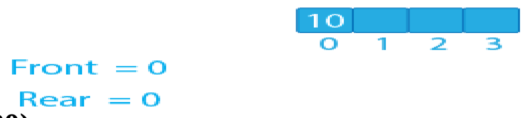
Example: We understood the concept of the queue. Now, we will learn about queue with the algorithm of a queue implementation with an example.

Initialize the front and rear variables with (-1).



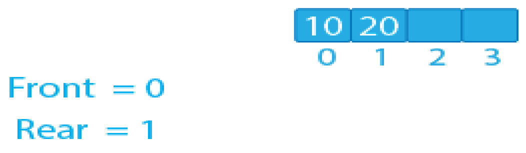
Operation 1: enqueue(10)

In this operation, we will insert a new value 10 at the rear position. Before this operation value of the rear was (-1). we will increment the value of the rear position by 1.So now, the new value of the rear will be 0.and the value 10 will be added at position 0.It is first element into the queue, that is why we will also increment the front position by 1 and the new value of the front will be also 0.



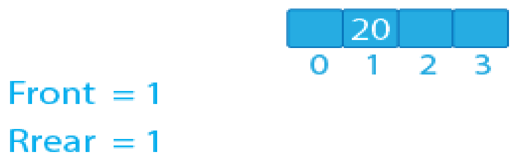
Operation 2: enqueue(20)

In this operation, we will insert a second value 20 at the rear position. Before this operation value of the rear was 0. we will increment the value of the rear position by 1.So now, the new value of the rear will be 1.and the value 20 will be added at position 1.So now, the new value of the front and rear will be 0 and 1 respectively.



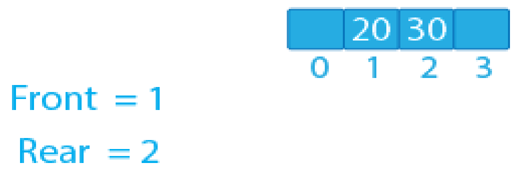
Operation 3: dequeue()

In this operation, we will delete the value 10 at the front position. Before this operation value of the front was 0 and the value 10 will be removed at position 0. After the deletion value, we will increment the value of the front position by 1. So now, the new value of the front and rear will be 1 and 1 respectively.



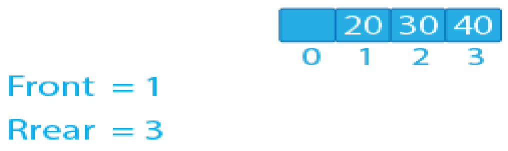
Operation 4: enqeue(30)

In this operation, we will insert a third value 30 at the rear position. Before this operation value of the rear was 1. we will increment the value of the rear position by 1.So now, the new value of the rear will be 2.and the value 30 will be added at position 2.So now, the new value of the front and rear will be 1 and 2 respectively.



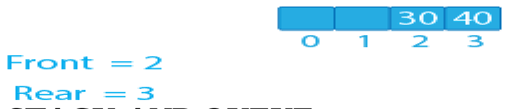
Operation 5: enqueue(40)

In this operation, we will insert a fourth value 40 at the rear position. Before this operation value of the rear was 2. we will increment the value of the rear position by 1.So now, the new value of the rear will be 3.and the value 40 will be added at position 3. So now, the new value of the front and rear will be 1 and 3 respectively.



Operation 6: dequeue()

In this operation, we will delete the value 20 at the front position. Before this operation value of the front was 1 and the value 20 will be removed at position 1. After the deletion value, we will increment the value of the front position by 1. So now, the new value of the front and rear will be 2 and 3 respectively.



DIFFERENCE BETWEEN STACK AND QUEUE

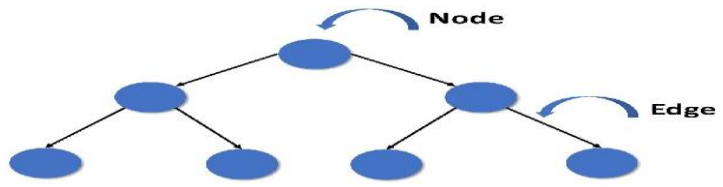
STACKS	**QUEUES**
1. A stack is a data structure that stores a collection of elements, with operations to push (add) and pop (remove) elements from the topof the stack.	1. A queue is a data structure that stores a collection of elements, with operations to enqueue (add) elements at the back of the queue, and dequeue (remove) elements from the front of the queue.
2. Stacks are based on the LIFO principle, i.e.,the element inserted at the last, is the first element to come out of the list.	2. Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
3. Insertion and deletion in stacks takes place only from one end of the list called the top.	3. Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
4. Insert operation is called push operation.	4. Insert operation is called enqueue operation.
5. Delete operation is called pop operation.	5. Delete operation is called dequeue operation.
6. Stacks are implemented using an array orlinked list data structure.	6. Queues are implemented using an array or linked listdata structure.

7. Stack does not have any types.	7. Queue is of three types – 1. Circular Queue 2. Priorityqueue 3. double-ended queue.

TREES

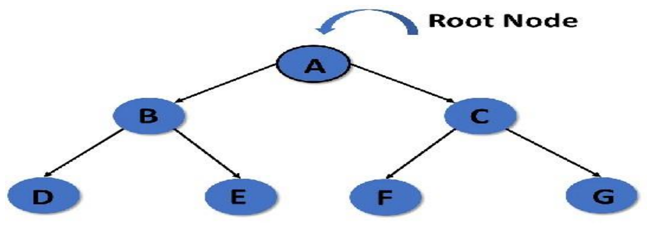
Tree: A tree is a A tree is a nonlinear data structure defined as a collection of nodes. Nodes represent value and nodes are connected by edges. It connects each node in the tree data structure using "edges", both directed and undirected.

The image below represents the tree data structure. The blue-colored circles depict the nodes of the tree and the black lines connecting each node with another are called edges.

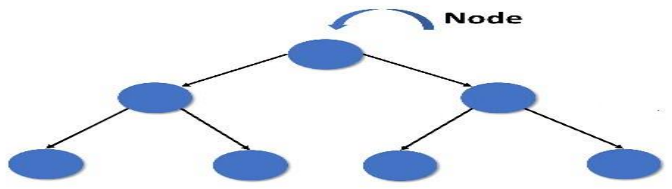


TREE TERMINOLOGIES

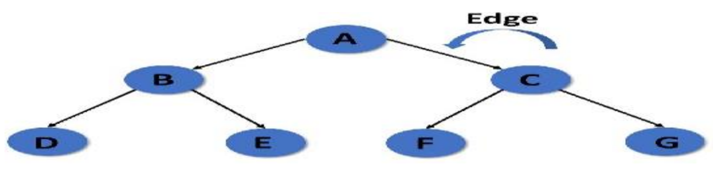
1. Root: In a tree data structure, the root is the first node of the tree. The root node is the initial node of the tree in data structures. In the tree data structure, there must be only one root node.



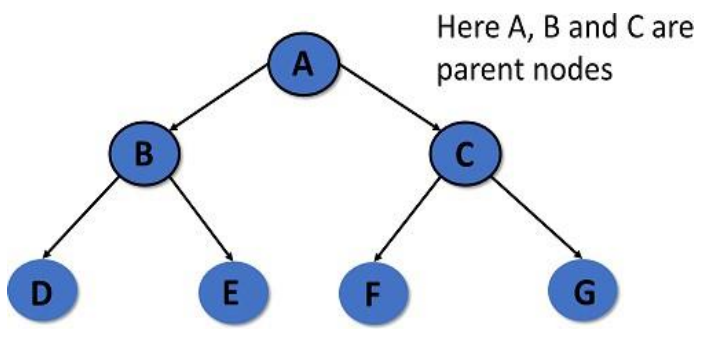
2. Node: A node is an entity that contains a key or value and pointers to its child nodes.



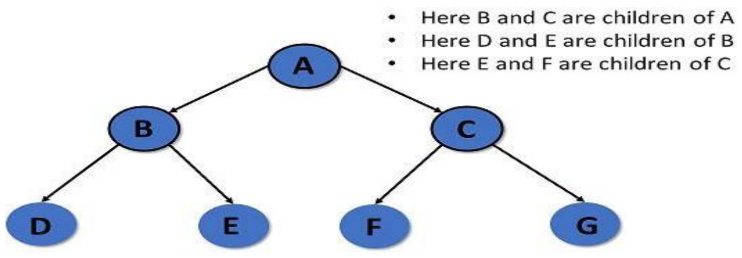
3. Edge : In a tree in data structures, the connecting link of any two nodes is called the Edge of the tree. In the tree data structure, N number of nodes connecting with N-1 number of edges.



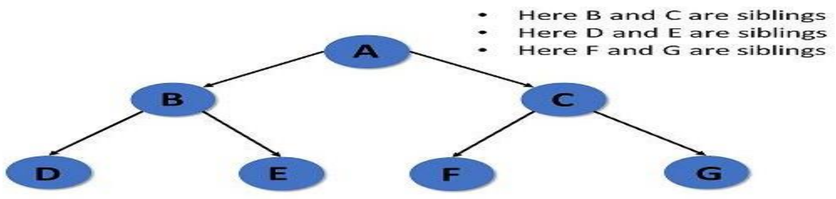
3. Parent : In the tree in data structures, the node that is the predecessor of any node is known as a parent node. The node which has one or more children is called as a parent node.



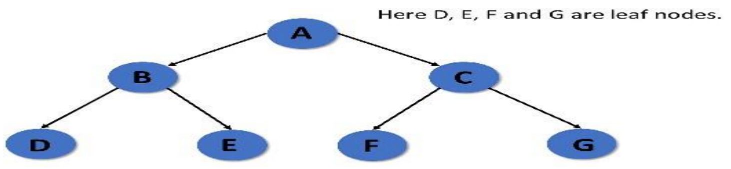
4. Child: The node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, every node except the root node is a child node.


Here B and C are children of A
Here D and E are children of B
Here E and F are children of C
A
B
C
D
E
F
G

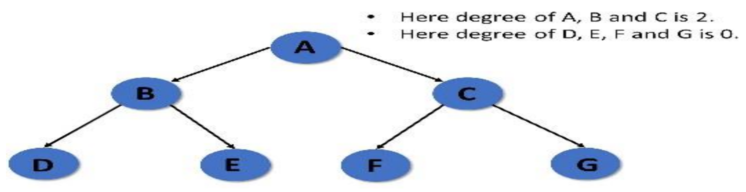
5. Siblings: In trees in the data structure, nodes that belong to the same parent are called siblings.


Here B and C are siblings
Here D and E are siblings
Here F and G are siblings
A
B
C
D
E
F
G

6. Leaf: In tree data structure, the node with no child, is known as a leaf node. In trees, leaf nodes are also called external nodes or terminal nodes.

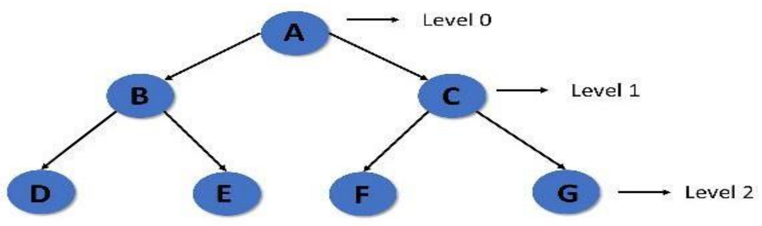


7. Degree: In the tree data structure, the total number of children of a node is called the degree of the node. The highest degree of the node among all the nodes in a tree is called the Degree of Tree.

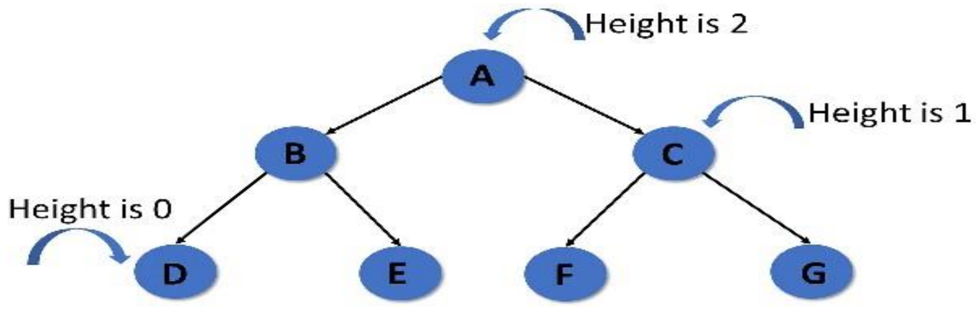


Here degree of A, B and C is 2.
Here degree of D, E, F and G is 0.
A
B
C
D
E
F
G

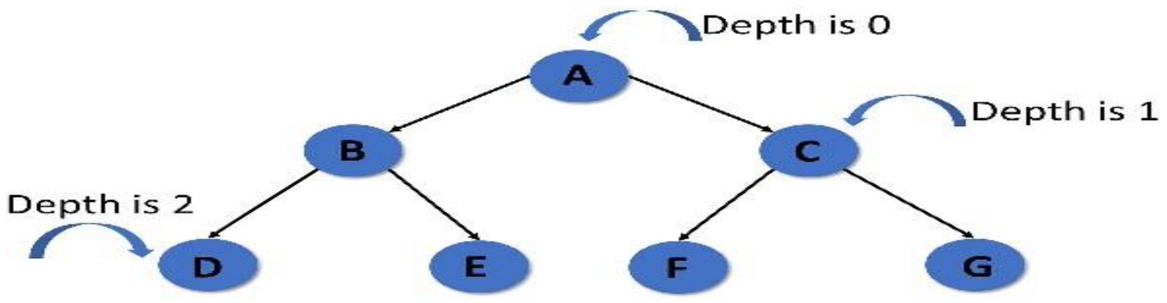
8. Level: In a tree, each step from top to bottom is called as level of a tree. The level count starts with 0 and increments by 1 at each level or step. In tree data structures, the root node is said to be at level 0, and the root node's children are at level 1, and the children of that node at level 1 will be level 2, and so on.

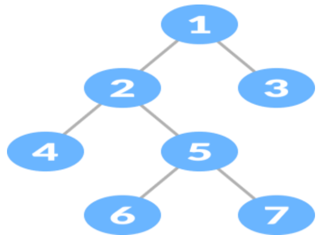


9. Height: In a tree data structure, the number of edges from the root node to the leaf node is known as the Height of that node. The tree height of all leaf nodes is 0.



10. Depth: The depth of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0





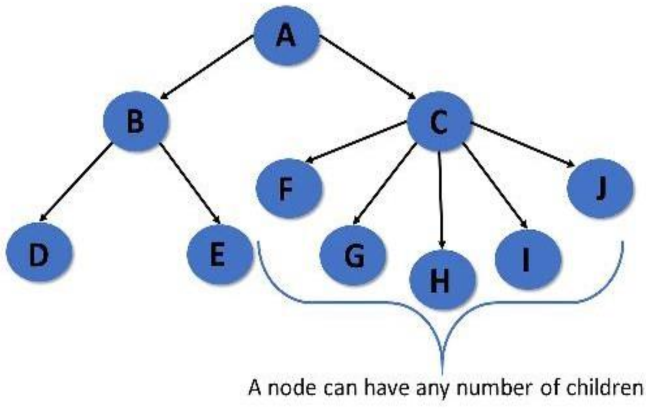
TYPES OF TREE IN DATA STRUCTURES: There are the different kinds of tree in data structures,

1. General Tree

The general tree is the type of tree where there are no constraints on the hierarchical structure.

Properties

- The general tree follows all properties of the tree data structure.
- A node can have any number of nodes.



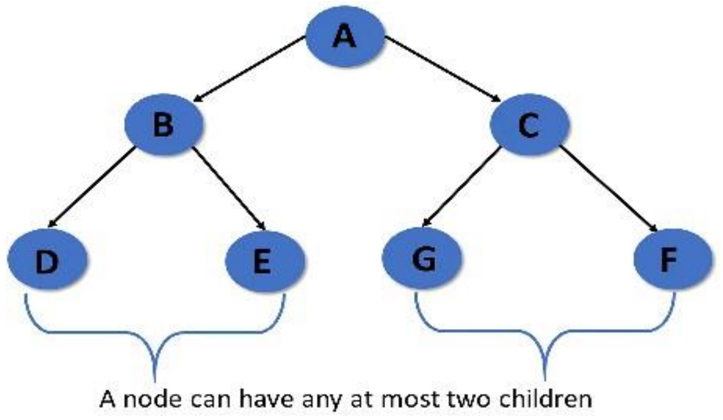
2. Binary Tree: A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

1. Data item
2. Address of left child
3. Address of right child

A binary tree has the following properties:

Properties

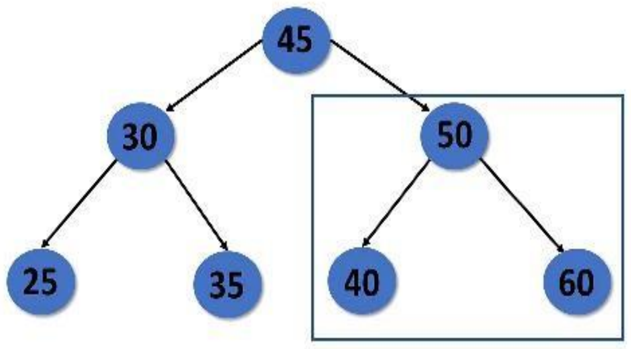
- Follows all properties of the tree data structure.
- Binary trees can have at most two child nodes.
- These two children are called the left child and the right child.



3. Binary Search Tree : It is a binary tree in which every node contains only smaller values in its left sub tree and onlylarger values are in its right sub tree.

Properties

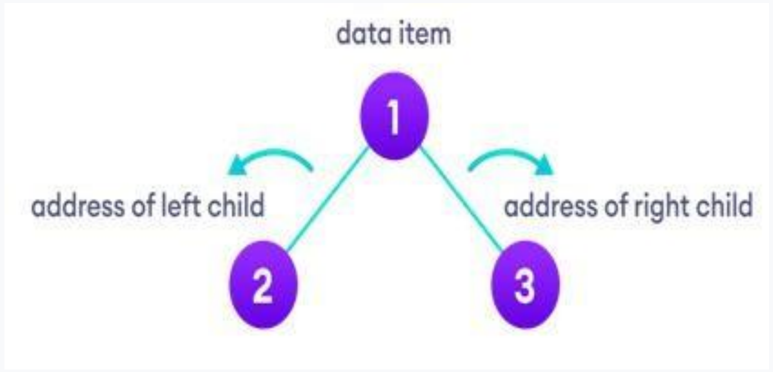
1. All nodes of left sub tree are less than the root node
2. All nodes of right sub tree are more than the root node
3. Both sub trees of each node are also BSTs i.e. they have the above two properties



Left node value<= root node <= right node value

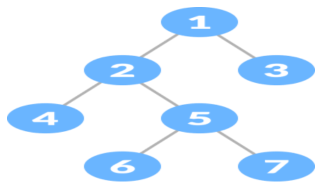
BINARY TREE: A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

1. Data item
2. Address of left child
3. Address of right child

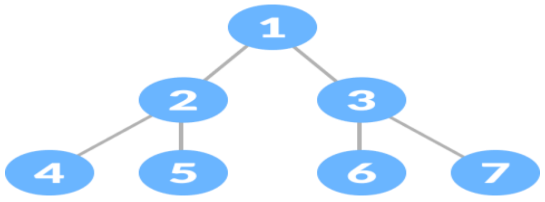


TYPES OF BINARY TREE: A binary tree contains following trees

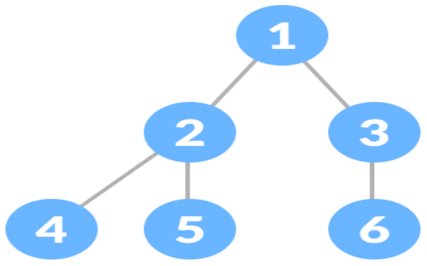
1. Full Binary Tree: A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



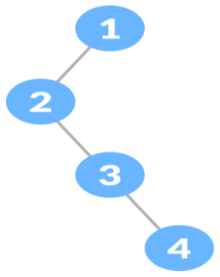
2. Perfect Binary Tree: A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



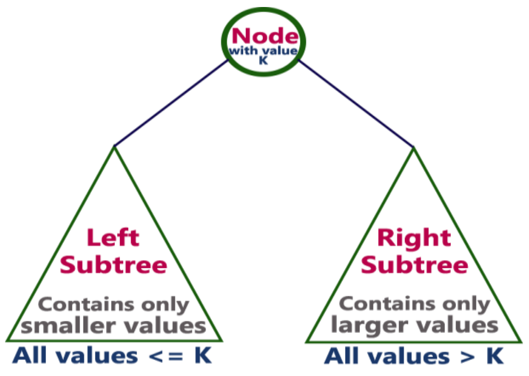
3. Complete Binary Tree: A binary tree is called complete binary tree when all the level of binary tree is completely filled except possibly the last level, which is filled from left side.



4. Degenerate or Pathological Tree: Degenerate Binary Tree is a Binary Tree where every parent node has only one child node. A degenerate or pathological tree having a single child either left or right.

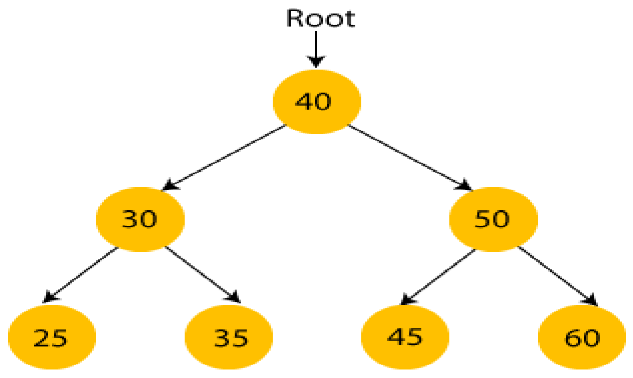


BINARY SEARCH TREE (BST): A Binary Search Tree (BST) is a tree in which every node contains only smaller values in its left sub tree and only larger values are in its right sub tree as shown in the following figure.



Example:

Consider the following binary search tree-



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node. Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Operations on binary search tree: The following operations are performed on binary search tree

1. Insertion
2. Deletion
3. Search(Traversing)

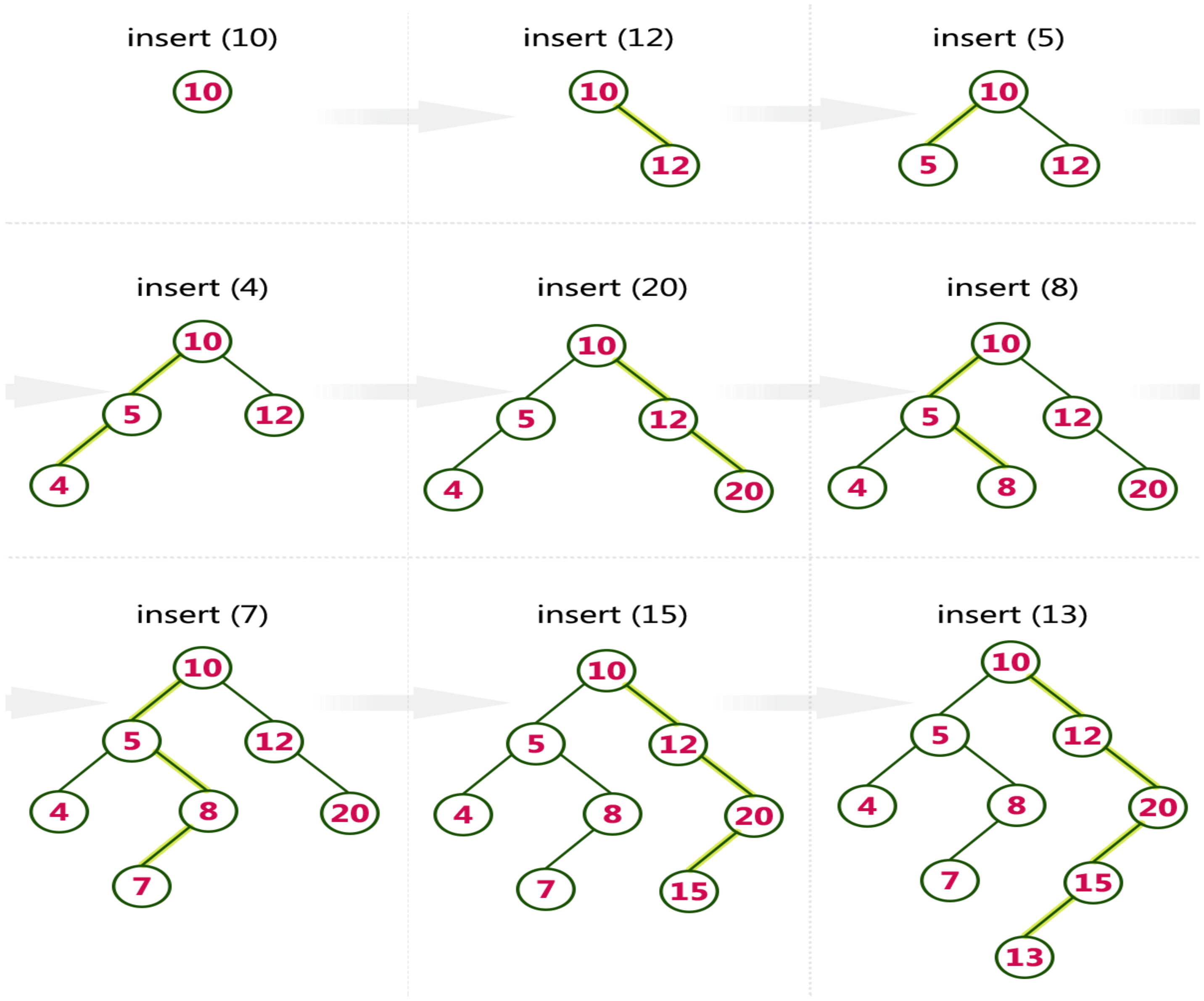
1. Insertion:

Example

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

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

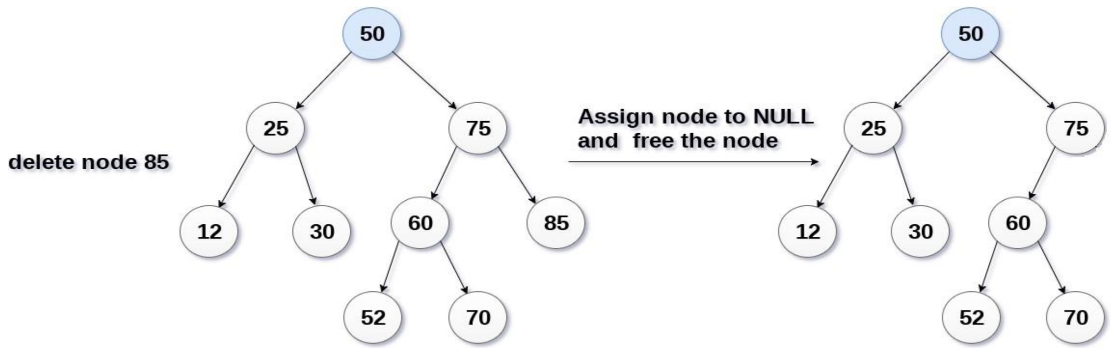
Above elements are inserted into a Binary Search Tree as follows...



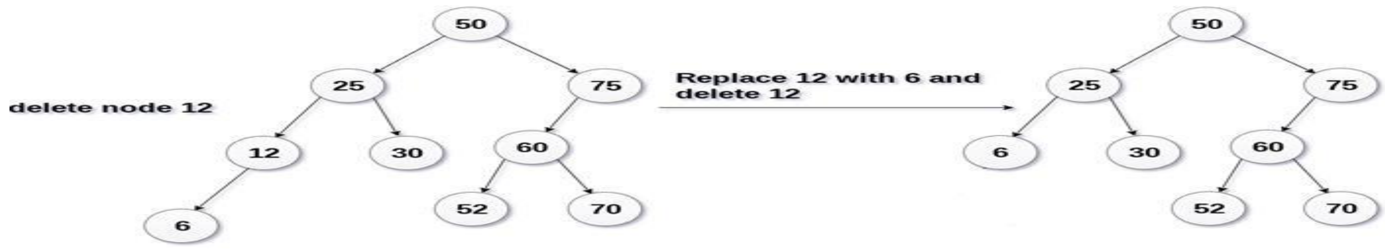
2. Deletion: Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

a. The node to be deleted is a leaf node
b. The node to be deleted has only one child.
c. The node to be deleted has two children.

a. The node to be deleted is a leaf node: If the node to be deleted from the tree has no child nodes, the node is simple deleted from the tree since it is a leaf node. In the following image, we are deleting the node 85, since the node is a leaf node.

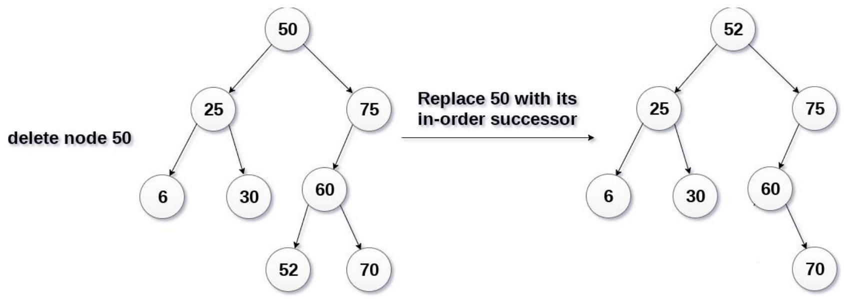


b. The node to be deleted has only one child: In this case, If the node to be deleted has a single child node, the target node is replaced from its child node and then the target node is simply deleted. In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



c. The node to be deleted has two children: It is a bit difficult compare to other two cases. However, the node which is to be deleted is replaced with its in-order successor or predecessor. In the following image, the node 50 is to be deleted which is the root node of the tree.

Replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



TREE TRAVERSAL TECHNIQUES:

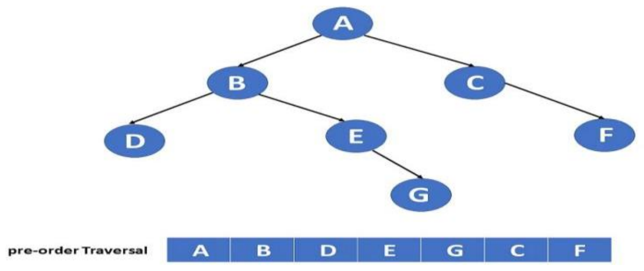
Tree Traversal: Tree traversals are nothing but visiting all the elements/nodes in the tree. There are 3 types of traversals. They are:

1. Pre-order
2. In-order

3. Post-order

1. Pre-order traversal: (R N, L, R): In pre-order traversal the root node is traversed first, then left sub-tree is traversed and then the right sub tree is traversed. The steps for traversing a tree in pre order traversal are:

a. Visit root
b. Visit left sub tree
c. Visit right sub tree

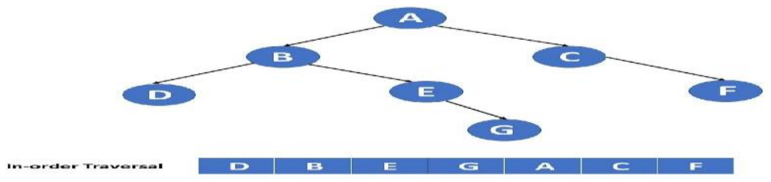


Example:

2. In ordered traversal: (L, RN, R): In ordered traversals left sub tree is traversed first. Then root is processed and then the right subtree is processed. The steps for traversing a tree in in order traversal are:

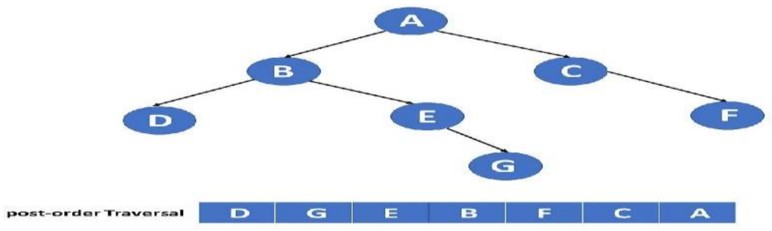
a. Visit left sub tree
b. Visit root
c. Visit right sub tree

Example:



3. Post order traversal: (L, R,RN): In post order traversal the left sub tree is traversed first and then the right sub tree is traversed and finally the root is traversed. The steps for traversing a tree in post order traversal are:

a. Visit left sub tree
b. Visit right sub tree
c. Visit root



Example:

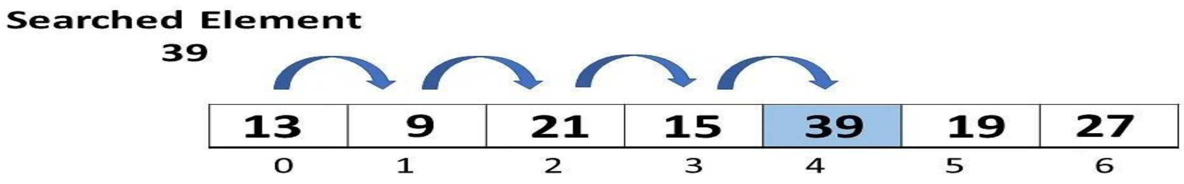
SEARCHING AND SORTING

Searching: Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

There are two popular search methods

1. Linear Search
2. Binary Search.

LINEAR SEARCH: Linear search is also called as sequential search algorithm**.** It is the simplest searching algorithm. Linear Search algorithm compares the search element to each element in the list and returns the location of the element if found otherwise, the algorithm returns NULL.

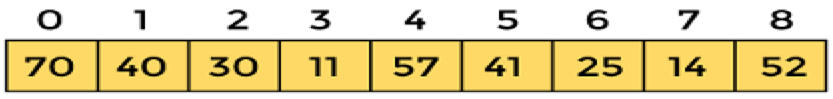


Method to use Linear Search

1. Start from the first element and compare each element with the search element.
2. If the element is found, return at which position element was found.
3. If the element is not found, return -1.

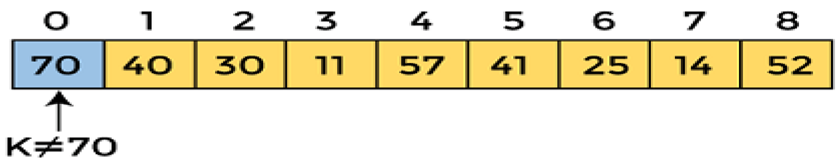
Working of Linear search: Now, let's see the working of the linear search Algorithm. To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

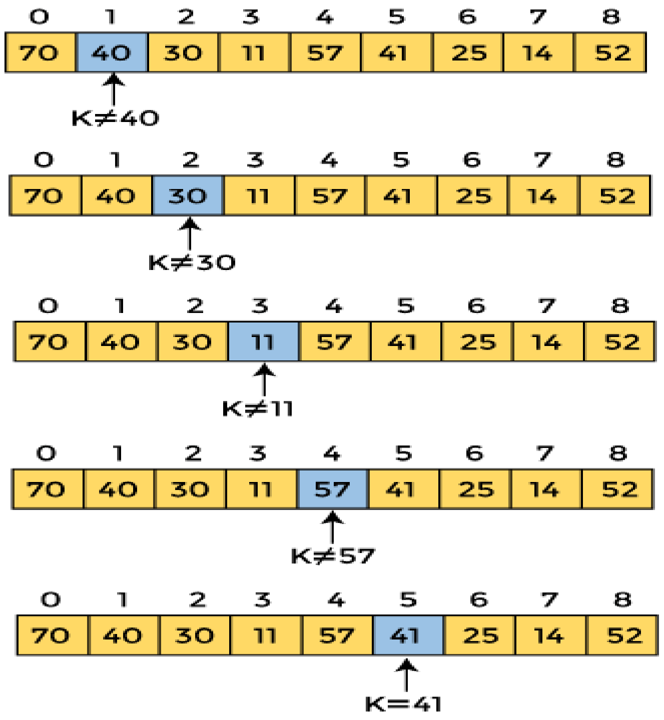


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.

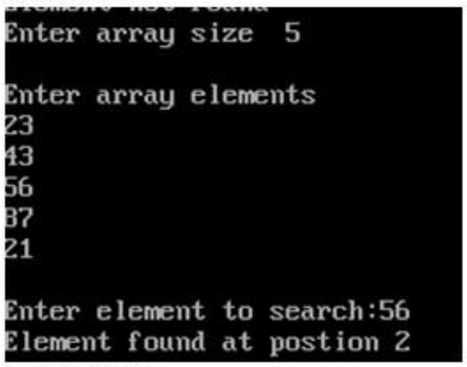


Now, the element to be searched is found. So algorithm will return the index of the element matched i.e. 5

/*Write a program on Linear Search using C */

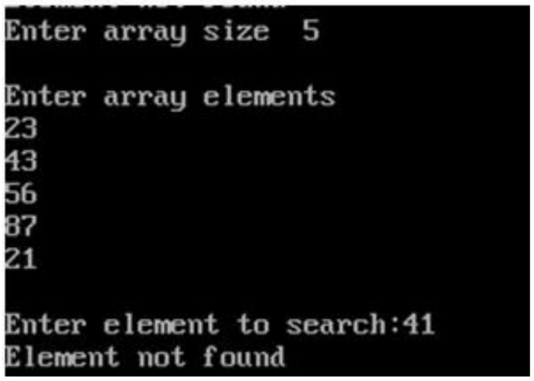
```
#include<stdio.h>
#include<conio.h>
void main()
      {
            int a[20],i,x,n;
            clrscr();
            printf("\nEnter array size ");
            scanf("%d",&n);
            printf("\nEnter array elements\n");
            for(i=0;i<n;++i)
            scanf("\n%d",&a[i]);
            printf("\nEnter element to search:");
            scanf("\n%d",&x);
            for(i=0;i<n;++i)
            if(a[i]==x)
            break;
            if(i<n)
            printf("Element found at position %d",i);
            else
            printf("Element not found");
            getch();
      }
```

OUTPUT: 1



```
Enter array size  5

Enter array elements
23
43
56
87
21

Enter element to search:56
Element found at postion 2
```

OUTPUT: 2



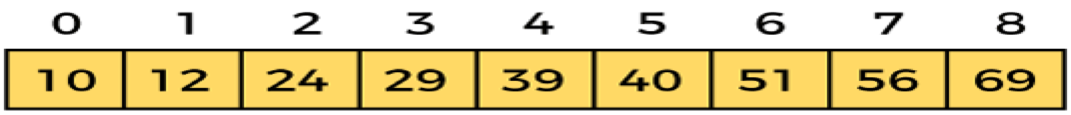
```
Enter array size  5

Enter array elements
23
43
56
87
21

Enter element to search:41
Element not found
```

2. BINARY SEARCH: Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array. A binary search technique works only on a sorted array, so an array must be sorted to apply binary search on the array. **Working of Binary search:** Now, let's see the working of the Binary Search Algorithm. To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of **Binary search with an example:** There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquers approach.

Let the elements of array are -



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

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

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

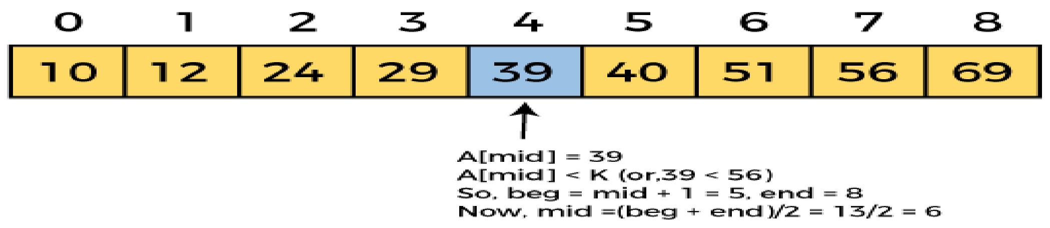
mid = (beg + end)/2

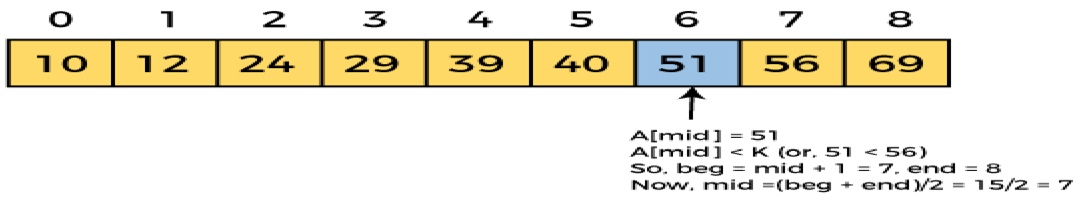
So, in the given array -

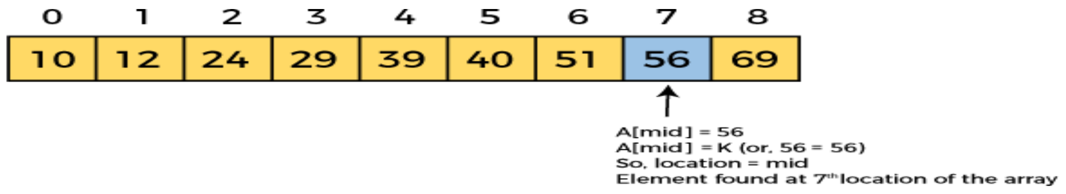
beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.





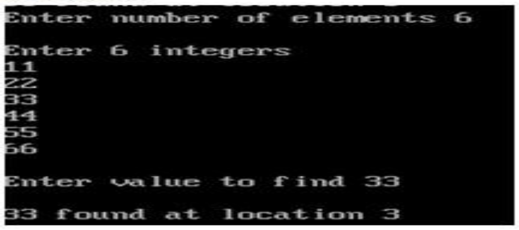


Now, the element to search is found. So algorithm will return the index of the element matched.

```
/*Write a program on Binary search using c*/
 #include <stdio.h>
 #include <conio.h>
 void main()
    {
          int i,low,high,mid,n,key,a[20];
          clrscr();
          printf("\nEnter number of elements ");
          scanf("%d",&n);
```

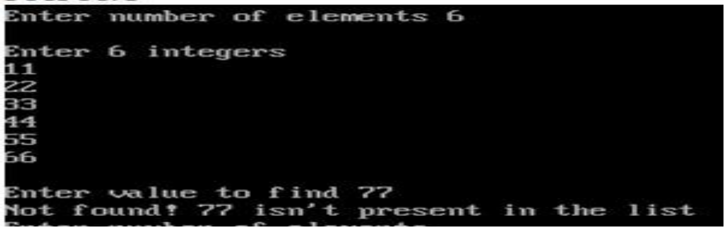
```
        printf("\nEnter %d integers",n);
        for(i=0;i<n;i++)
        scanf("\n%d",&a[i]);
        printf("\nEnter value to find ");
        scanf("%d",&key);
        low=0;
        high=n-1;
        mid=(low+high)/2;
        while(low<=high)
        {
        if(a[mid]<key)
        low=mid+1;
        else if(a[mid]== key)
        {
        printf("\n%d found at location %d",key,mid+1);
        break;
        }
        else
        high=mid-1
        mid=(low+high)/2;
        }
        if(low>high)
            printf("Not found! %d isn't present in the list", key);
        getch();
}
```

OUTPUT: 1



```
Enter number of elements 6

Enter 6 integers
11
22
33
44
55
66

Enter value to find 33

33 found at location 3
```

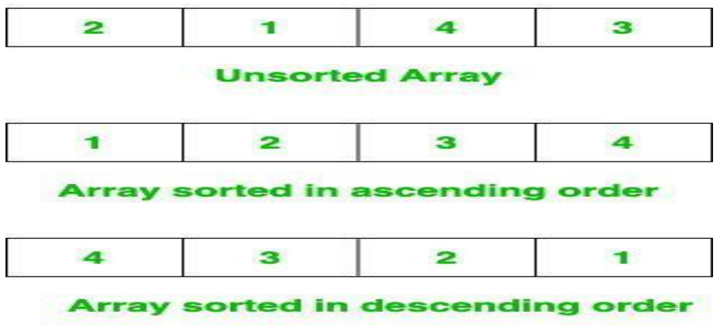
OUTPUT: 2



```
Enter number of elements 6

Enter 6 integers
11
22
33
44
55
66

Enter value to find 77
Not found! 77 isn't present in the list
```

SORTING: Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. Sorting techniques are used to arrange data(mostly numerical) in an ascending or descending order.

Example



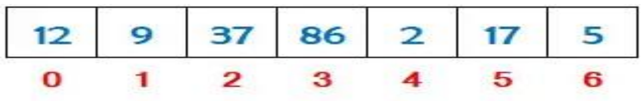
2 1 4 3
Unsorted Array
1 2 3 4
Array sorted in ascending order
4 3 2 1
Array sorted in descending order

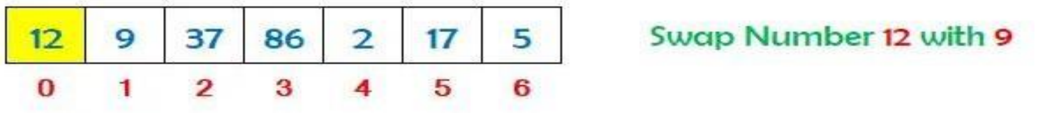
Sorting can be performed using several techniques or methods, as follows:

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Quick sort
5. Merge sort

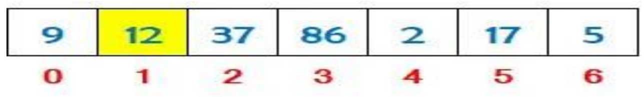
1. BUBBLE SORT: Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Working procedure of Bubble sort: Suppose we have seven numbers stored in an array as shown below.

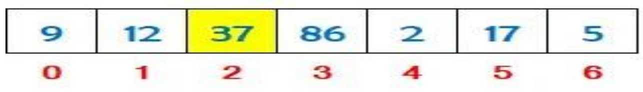




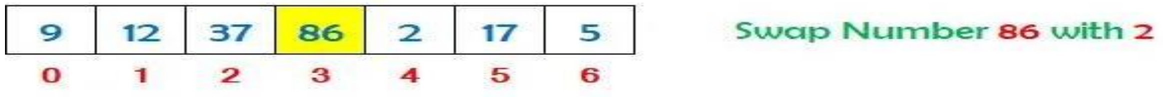
Step 1: Take the 1st number 12 and compare it with the 2nd number 9. 12 is greater than 9 so swap both the numbers.



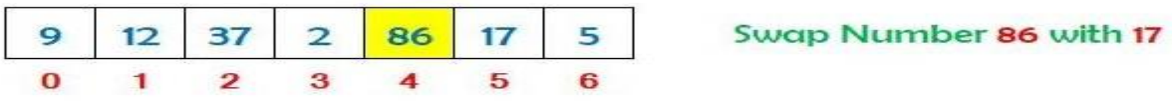
Step 2: Now take the 2nd number 12 and compare it with the 3rd number 37. 12 is not greater than 37 so leave it.



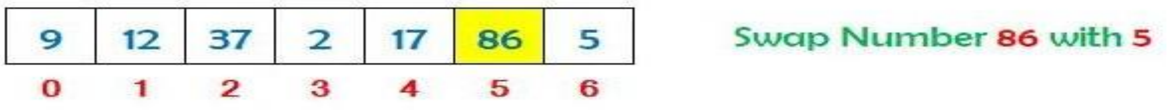
Step 3: Now take the 3rd number 37 and compare it with the 4th number 86. 37 is not greater than 86 so leave it.



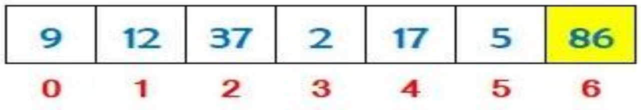
Step 4: Now take the 4th number 86 and compare it with the 5th number 2. 86 is greater than 2 so swap both the numbers.



Step 5: Now take the 5th number 86 and compare it with the 6th number 17. 86 is greater than 17 so swap both the numbers.



Step 6: Now take the 6th number 86 and compare it with the 7th number 5. 86 is greater than 5 so swap both the numbers.



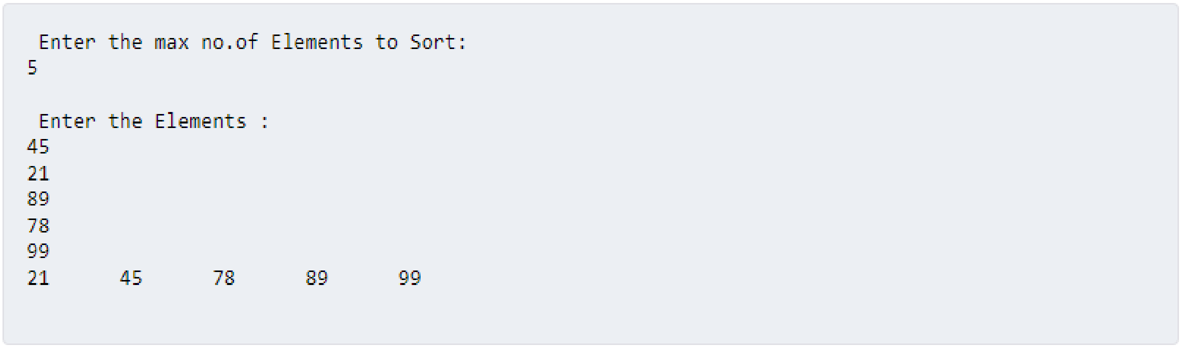
So we can see that the largest number has shifted towards the end of the array. To arrange the other numbers also, we have to repeat the above process five times more from index 0 to index 4. If there are ten numbers in an array, we have to repeat the above process nine times to sort the entire array.

/* Write a program on Bubble sort using C */

```
#include<stdio.h>
#include<conio.h>
void main()
     {
         int a[10],i,j,temp,n;
         printf("\n Enter the max no.of Elements to Sort: \n");
         scanf("%d",&n);
         printf("\n Enter the Elements : \n");
         for(i=0; i<n; i++)
           {
```

```
                    scanf("%d",&a[i]);
               }
          for(i=0; i<n; i++)
               {
                    for(j=i+1; j<n; j++)
                      {
                         if(a[i]>a[j])
                           {
                                temp=a[i];
                                a[i]=a[j];
                                a[j]=temp;
                           }
                      }
               }
        for(i=0; i<n; i++)
             {
                  printf("%d\t",a[i]);
             }
       getch();
 }
```

Output:

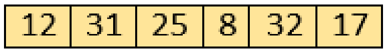


```
 Enter the max no.of Elements to Sort:
5

 Enter the Elements :
45
21
89
78
99
21      45      78      89      99
```

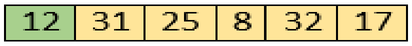
2. INSERTION SORT: Insertion sort algorithm is a basic sorting algorithm that sequentially sorts each item in the final sorted array or list.

Working of Insertion sort Algorithm: Now, let's see the working of the insertion sort Algorithm. To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

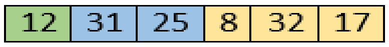
Let the elements of array are -

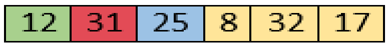


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



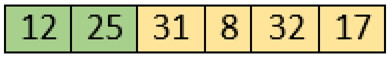
Now, move to the next two elements and compare them.



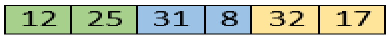


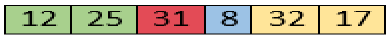
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

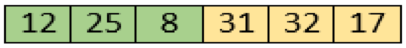


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

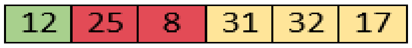




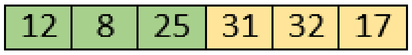
Both 31 and 8 are not sorted. So, swap them.

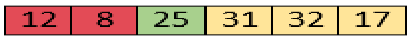


After swapping, elements 25 and 8 are unsorted.



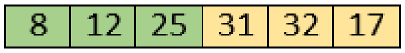
So, swap them.



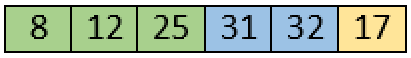


Now, elements 12 and 8 are unsorted.

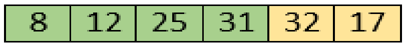
So, swap them too.



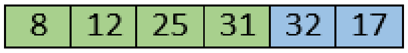
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



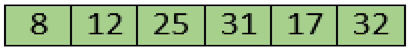
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

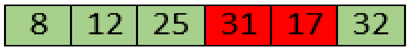


Move to the next elements that are 32 and 17.

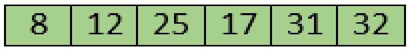


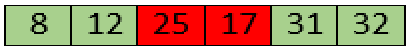
17 is smaller than 32. So, swap them.



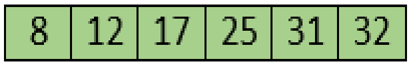


Swapping makes 31 and 17 unsorted. So, swap them too.





Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

/* Write a program on Insertion sort using C */

```
#include <stdio.h>
void main(void)
  {
       int n, i, j, temp;
       int arr[64];
       printf("Enter number of elements\n");
       scanf("%d", &n);
       printf("Enter %d integers\n", n);
       for (i = 0; i < n; i++)
         {
             scanf("%d", &arr[i]);
         }
       for (i = 1; i < n; i++)
        {
```

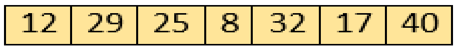
```
                 j = i;
                while (j > 0 && arr[j - 1] > arr[j])
                {
                       temp = arr[j];
                       arr[j] = arr[j - 1];
                       arr[j - 1] = temp;
                       j--;
                  }
            }
                  printf("Sorted list in ascending order:\n");
                  for (i = 0; i < n; i++)
                     {
                          printf("%d\n", arr[i]);
                     }
        getch();
    }OUTPUT:
```

```
Enter number of elements
5
Enter 5 integers
3 7 2 9 5
Sorted list in ascending order:
2
3
5
7
9
```

3. SELECTION SORT: Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Working of Selection sort Algorithm: Now, let's see the working of the Selection sort Algorithm. To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

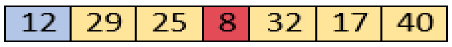
Let the elements of array are -



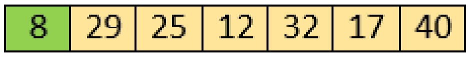
12	29	25	8	32	17	40

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

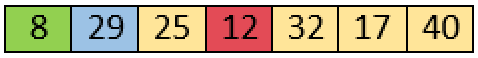
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.



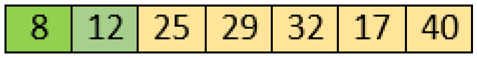
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.



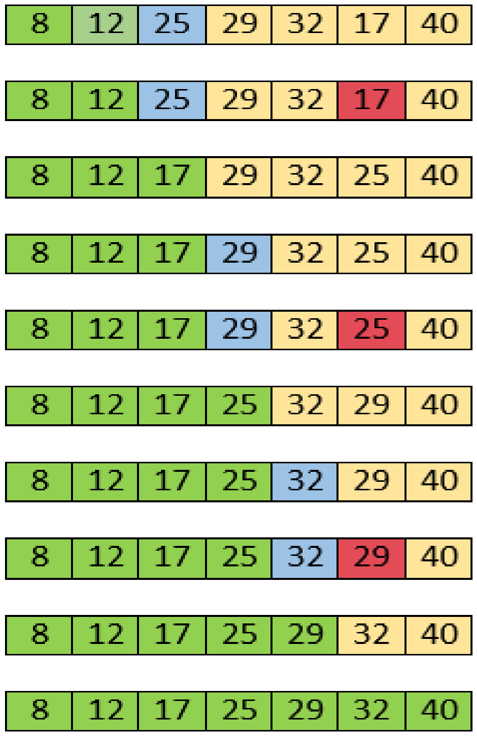
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.



Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

4. QUICK SORT: Quick Sort is a Sorting algorithm, which uses Divide and Conquer Approach to perform sorting. Quick Sort is the most efficient algorithm among all other sorting algorithms, as sorting can be done in O(n*logn) time.

Divide and conquer is a technique of breaking down the algorithms into sub problems, then solving the sub problems, and combining the results back together to solve the original problem.

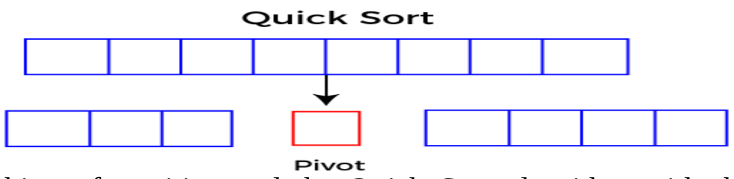
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two sub arrays with Quick sort.

Working of Quick Sort Algorithm

Quick sort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

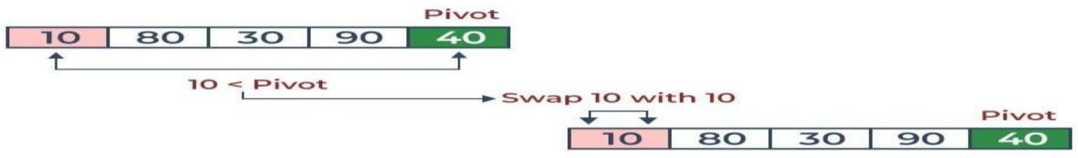


Let us understand the working of partition and the Quick Sort algorithm with the help of the following example:

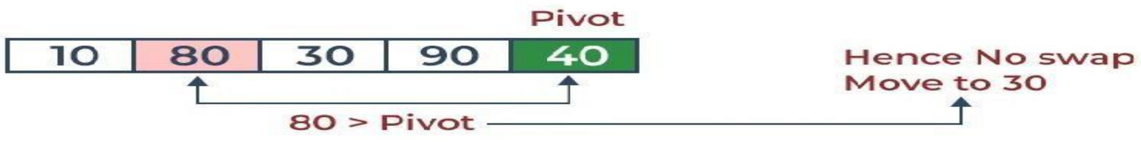
Consider: arr[] = {10, 80, 30, 90, 40}.

In the given array, we consider the right most element as pivot.

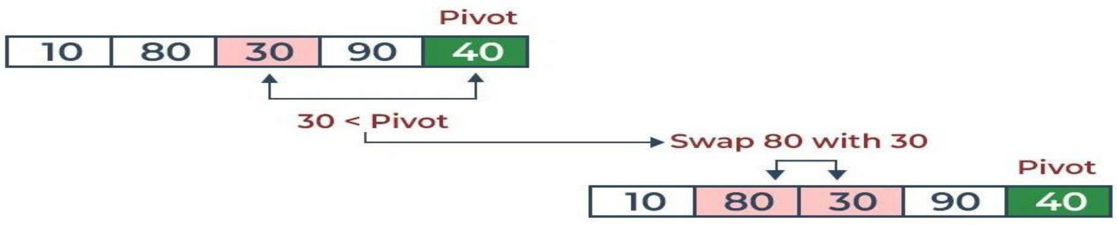
- Compare 10 with the pivot and as it is less than pivot arrange it accordingly.



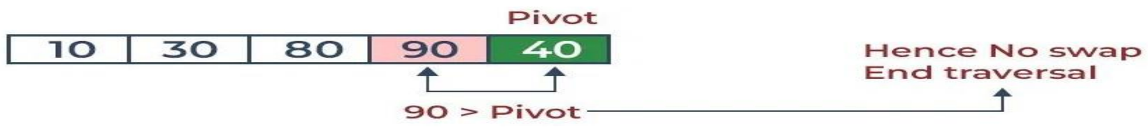
- Compare 80 with the pivot. It is greater than pivot.



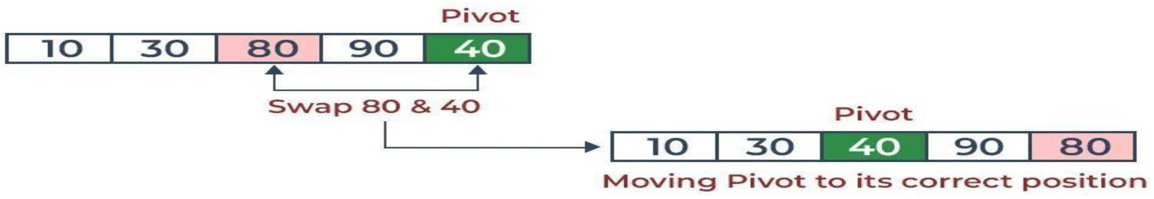
- Compare 30 with pivot. It is less than pivot so arrange it accordingly.



- Compare 90 with the pivot. It is greater than the pivot.



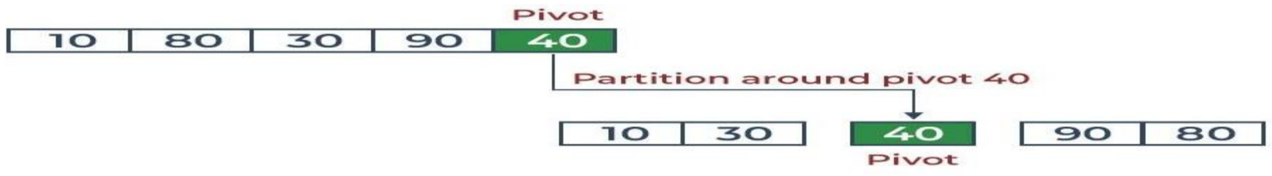
- Arrange the pivot in its correct position.



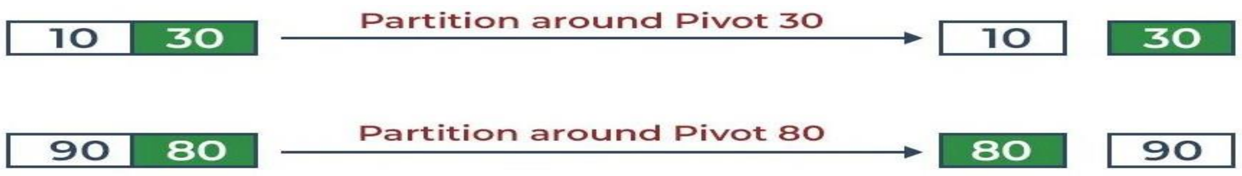
As the partition process is done recursively, it keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted.

Follow the below images to understand how the recursive implementation of the partition algorithm helps to sort the array.

- Initial partition on the main array:



- Partitioning of the subarrays:



```
/* Write a program on Insertion sort using C */
#include <stdio.h>
 void quick_sort (int [], int, int);
 int main()
      {
              int array[50];
              int size, i;
              printf("Enter the number of elements: ");
              scanf("%d", &size);
```

```
                printf("Enter %d numbers : ",size);
                for (i = 0; i < size; i++)
                  {
                        scanf("%d", &array[i]);
                  }
            quick_sort(array, 0, size - 1);
            printf("Quick Sorted List \n");
            for (i = 0; i < size; i++)
              {
                printf("%d ", array[i]);
              }
                printf("\n");
          return 0;
   }
void quick_sort(int array[], int low, int high)
      {
                int pivot, i, j, temp;
                if (low < high)
                 {
                   pivot = low;
                   i = low;
                   j = high;
               while (i < j)
                  {
                     while (array[i] <= array[pivot] && i <= high)
                       {
                          i++;
                       }
                while (array[j] > array[pivot] && j >= low)
                   {
                         j--;
                   }
                if (i < j)
                {
                   temp = array[i];
                   array[i] = array[j];
                 array[j] = temp;
              }
          }
      temp = array[j];
      array[j] = array[pivot];
      array[pivot] = temp;
      quick_sort(array, low, j - 1);
      quick_sort(array, j + 1, high);
   }
}
```

```
Enter the number of elements: 8
Enter 8 numbers : 87 39 81 42 73 49 61 53
Quick Sorted List
39 42 49 53 61 73 81 87

Enter the number of elements: 6
Enter 6 numbers : 302 202 504 104 893 404
Quick Sorted List
104 202 302 404 504 893
```

OUTPUT:

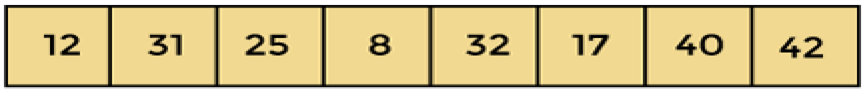
5. MERGE SORT: Merge sort is similar to the quick sort algorithm as it uses the divide and conquer based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Working of Merge sort Algorithm

Now, let's see the working of merge sort Algorithm.

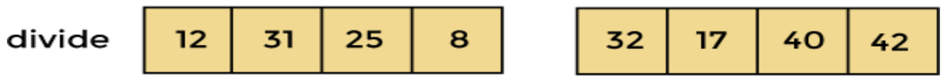
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

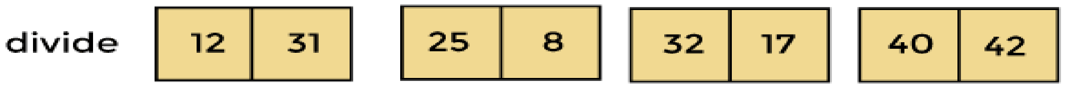


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

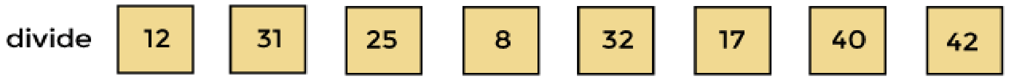
As there are eight elements in the given array, so it is divided into two arrays of size 4.



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



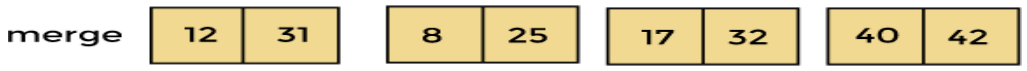
Now, again divide these arrays to get the atomic value that cannot be further divided.



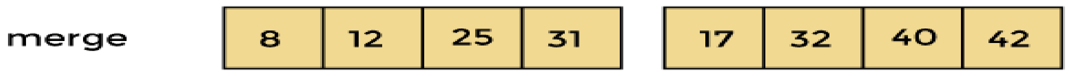
Now, combine them in the same manner they were broken.

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

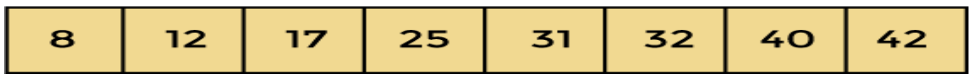
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



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



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



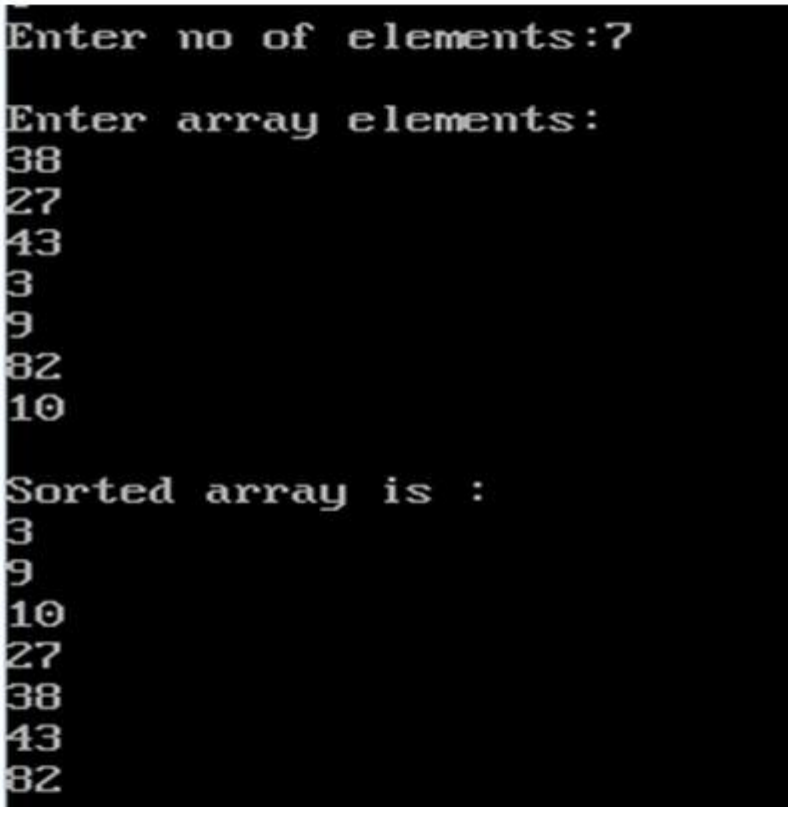
Now, the array is completely sorted.

/*Write a program on Merge Sort using C */

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
void main()
{
    int a[30],n,i;
    printf("\nEnter no of elements:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++) scanf("%d",&a[i]);
    mergesort(a,0,n-1);
    printf("\nSorted array is :");
    for(i=0;i<n;i++)
    printf("\n%d ",a[i]);
    getch();
}
void mergesort(int a[],int i,int j)
{
    int mid;
    if(i<j)
    {
        mid=(i+j)/2;
        mergesort(a,i,mid); mergesort(a,mid+1,j);
        merge(a,i,mid,mid+1,j);
    }
}
void merge(int a[],int i1,int j1,int i2,int j2)
{
    int temp[50];
    int  i,j,k; i=i1;
    j=i2;k=0;
    while(i<=j1 && j<=j2)
```

```
        {
                if(a[i]<a[j])
                temp[k++]=a[i++];else
                temp[k++]=a[j++];
        }
while(i<=j1)
 temp[k++]=a[i++];
while(j<=j2)
temp[k++]=a[j++];
for(i=i1,j=0;i<=j2;i++,j++)
a[i]=temp[j];
}
```

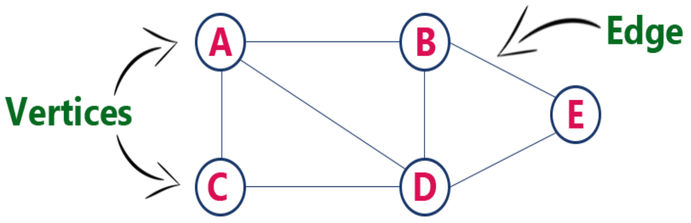
OUTPUT:



```
Enter no of elements:7

Enter array elements:
38
27
43
3
9
82
10

Sorted array is :
3
9
10
27
38
43
82
```

GRAPHS

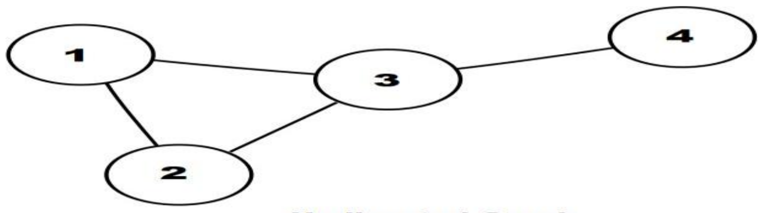
GRAPH: A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges(E). The graph



is denoted by G(E, V).

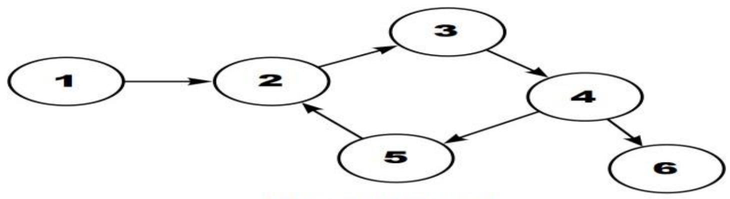
TYPES OF GRAPHS: The most common types of graphs in the data structure are below,

1. Undirected: A graph in which all the edges are bi-directional. The edges do not point in a specific direction.



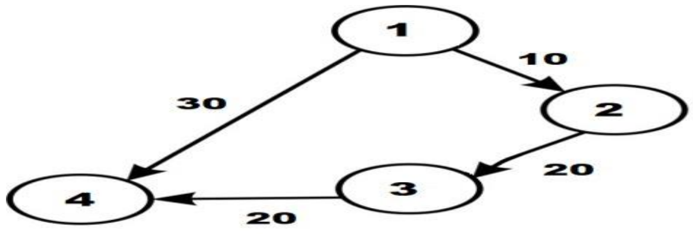
Undirected Graph

2. Directed: A graph in which all the edges are uni-directional. The edges point in a single direction.



Directed Graph

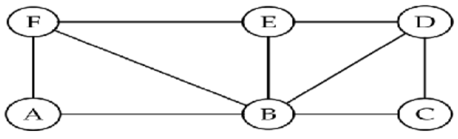
3. Weighted Graph: A graph with a value associated with every edge. The values corresponding to the edges are called weights. A value in a weighted graph can represent quantities such as cost, distance, and time, depending on the graph. We typically use weighted graphs in modelling



Weighted Graph

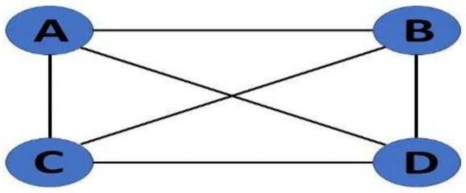
computer networks.

4. Unweighted Graph: A graph with no value or weight associated with the edge. All the graphs are unweighted by default unless there is a value associated.

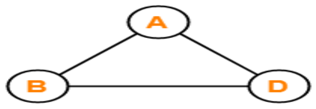


5. Null Graph: If a graph with no edges is called null graph (or) totally disconnected graph

6. Connected Graph: A connected graph has a path between every pair of vertices. In other words, there are no unreachable vertices. A disconnected graph is a graph that is not connected.



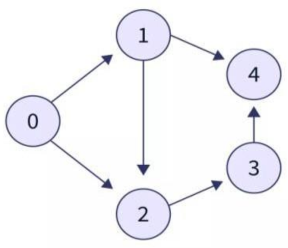
7. Complete Graph: A graph in which exactly one edge is present between every pair of vertices is called as a complete graph.

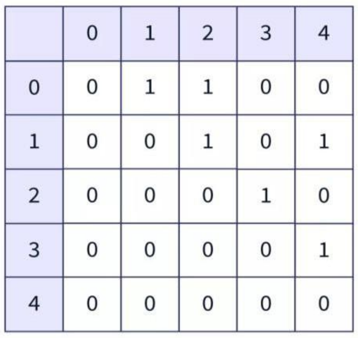


REPRESENTATIONS OF GRAPHS: The following two are the most commonly used representations of a graph.

1. **Sequential representation** (or, Adjacency matrix representation)
2. **Linked list representation** (or, Adjacency list representation)

1. Sequential representation (or, Adjacency matrix representation): In this type of representation, we create a 2D array, and each cell (i,j) represents if there is an edge that exists between node i and node j. If the value at cell (i,j) is 1, then there is an edge between i and j, else if the value is 0, then there is no edge. It is used to indicate which nodes are adjacent to one another. I.e., do any edges connect nodes in a graph.





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

Pros and Cons of the Adjacency Matrix

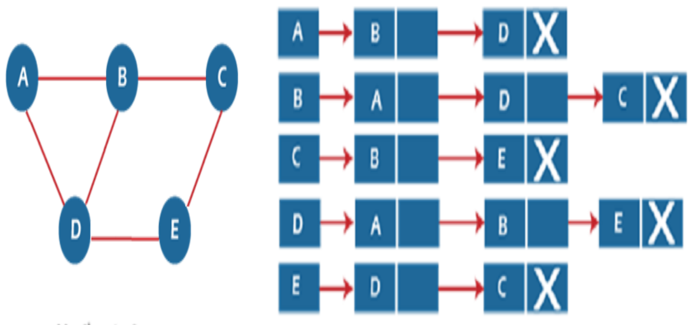
Pros: Representation is simpler to implement and adhere to.

Cons: It takes a lot of space and time to visit all of a vertex's neighbors; we have to traverse all of the vertices in the graph, which takes a long time.

2. Linked list representation An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

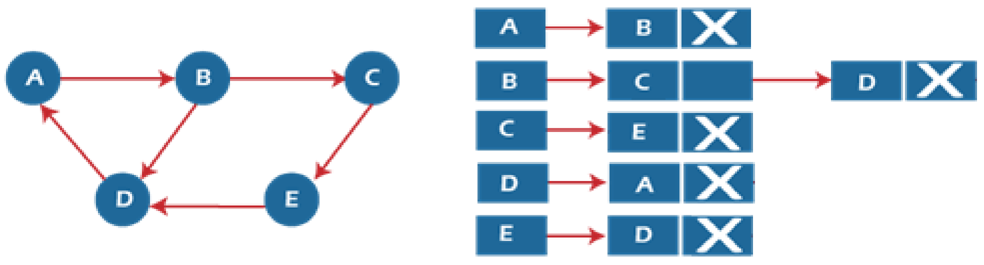
In the above figure, we can see that there is a linked list or adjacency list for every node of the graph.


Undirected Graph Adjacency List

From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

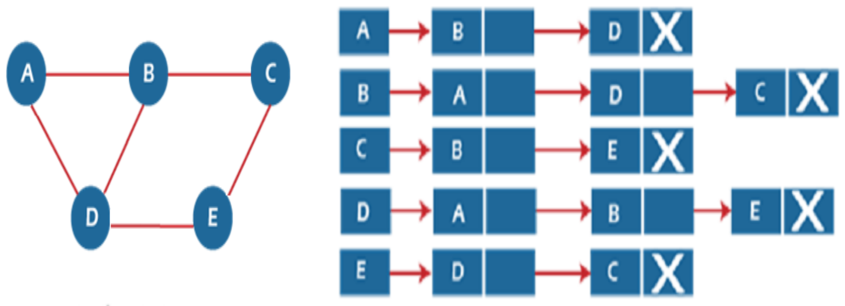

Directed Graph Adjacency List

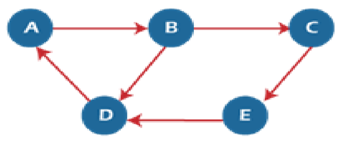
Now, consider the directed graph, and let's see the adjacency list representation of that graph.

For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

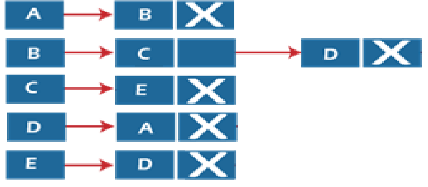
IMPLEMENTATION OF GRAPH USING LINKED LIST: An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.


Undirected Graph Adjacency List



Directed Graph



Adjacency List

In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

n adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph. For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

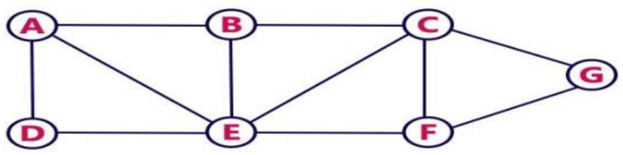
GRAPHS TRAVERSAL TECHNIQUES: Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. There are two graph traversal techniques and they are as follows...

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

1. BFS (Breadth First Search): The Breadth First Search (BFS) algorithm is used to search a graph data structure. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

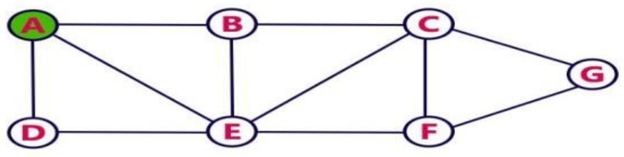
Example:

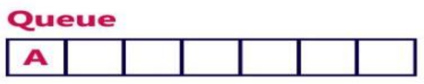
Consider the following example graph to perform BFS traversal



Step 1:

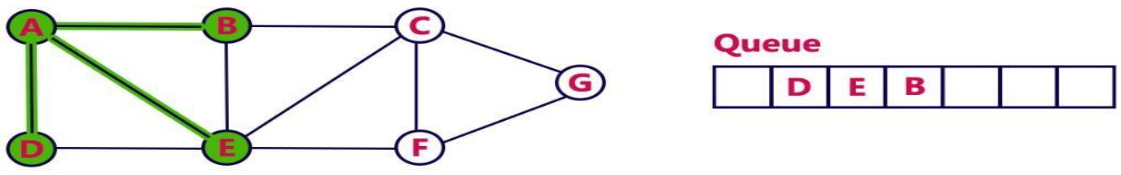
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.





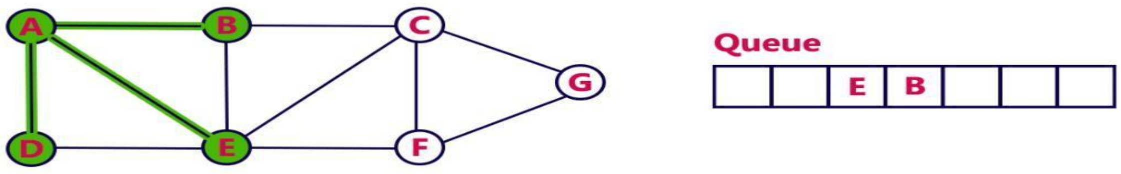
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



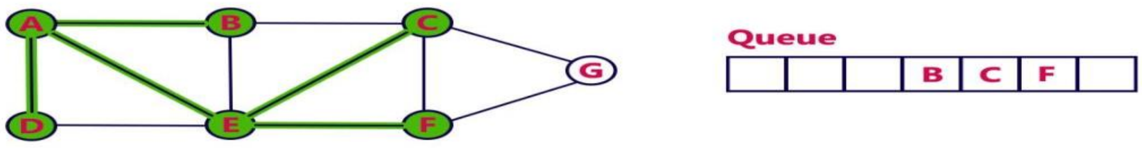
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



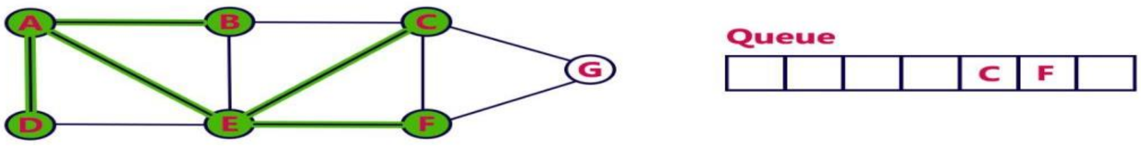
Step 4:

- 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.



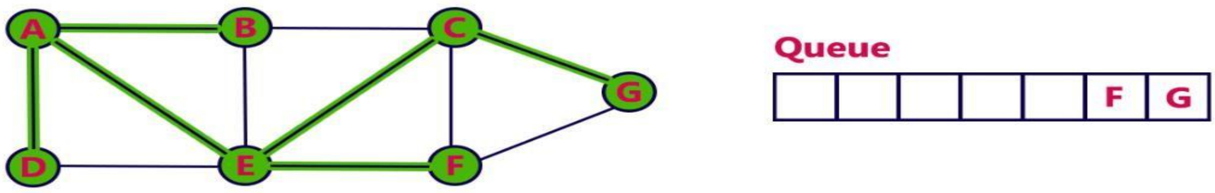
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



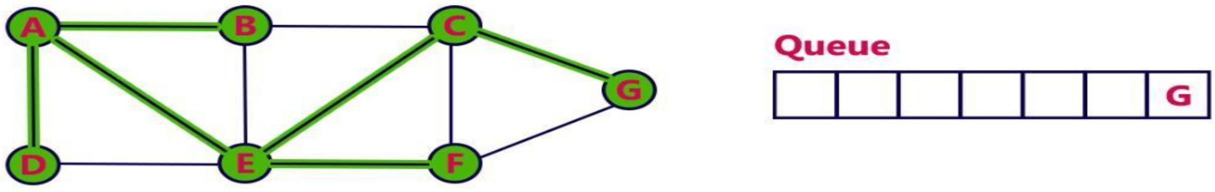
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



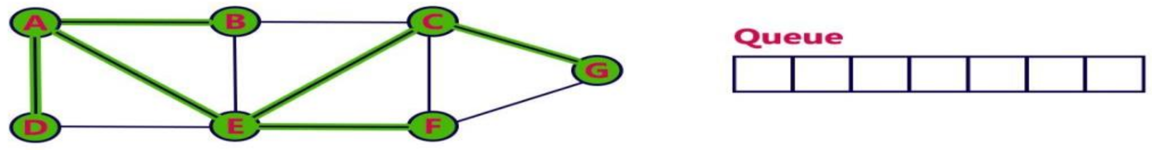
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

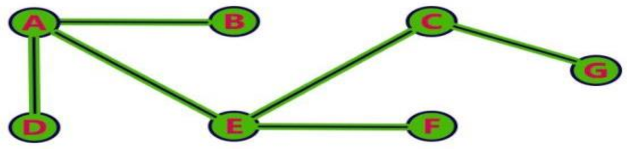


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



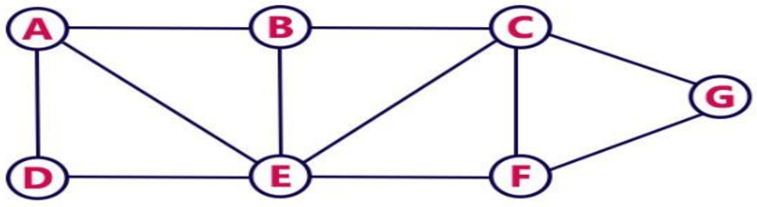
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



2. DFS (Depth First Search): Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (in the case of a graph, you can use any random node as the root node) and examines each branch as far as possible before backtracking.

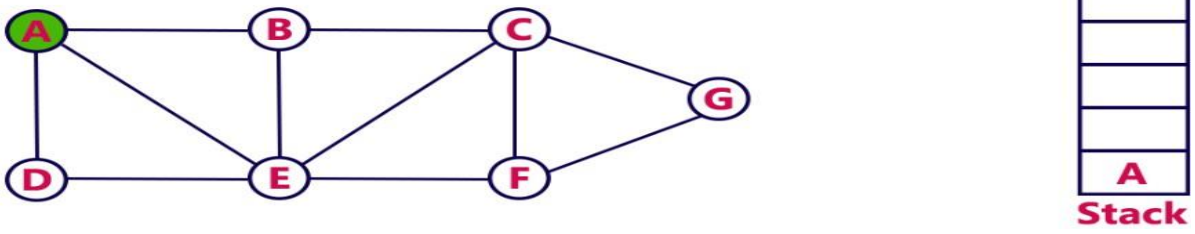
Example

Consider the following example graph to perform DFS traversal



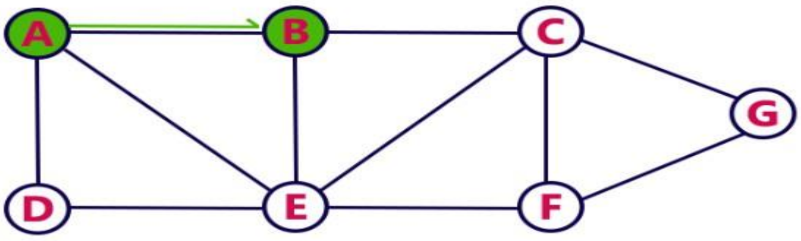
Step 1:

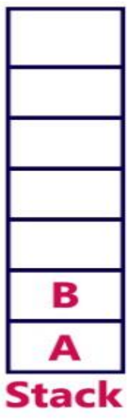
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



Step 2:

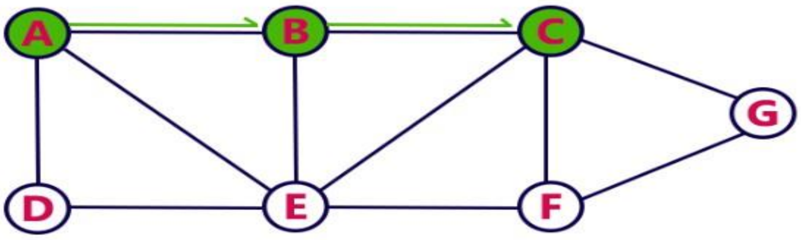
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

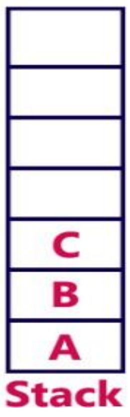




Step 3:

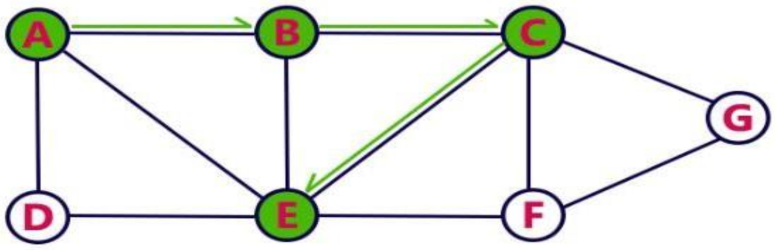
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

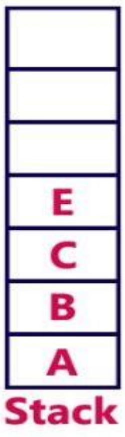




Step 4:

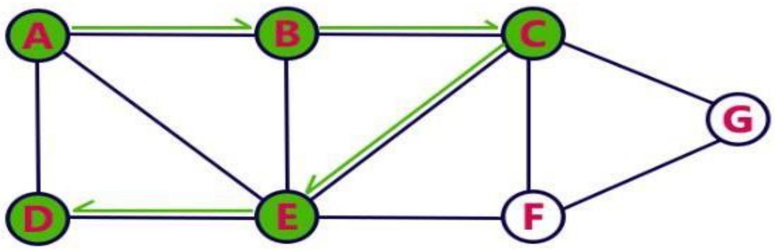
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

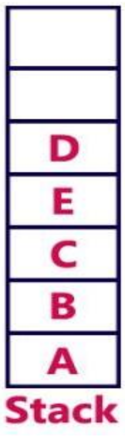




Step 5:

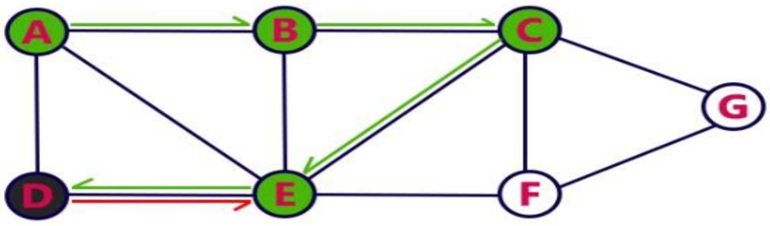
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

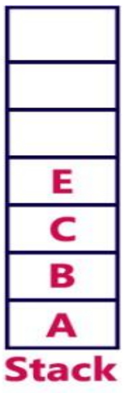




Step 6:

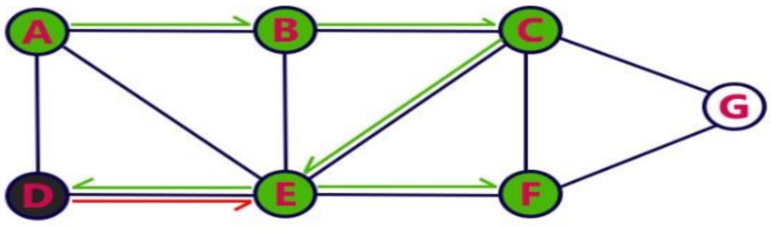
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

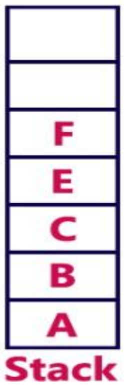




Step 7:

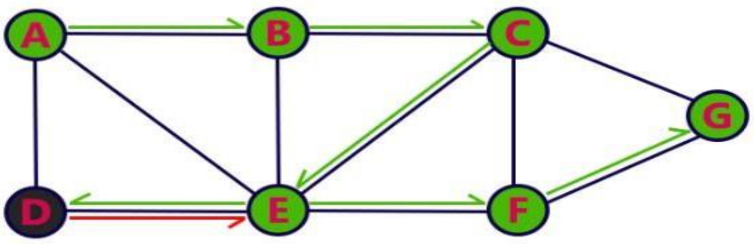
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

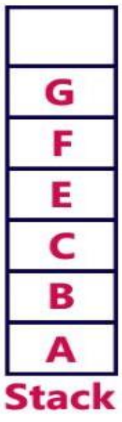




Step 8:

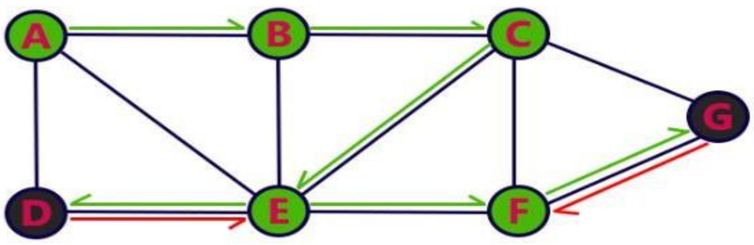
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

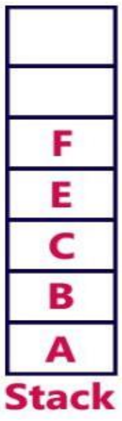




Step 9:

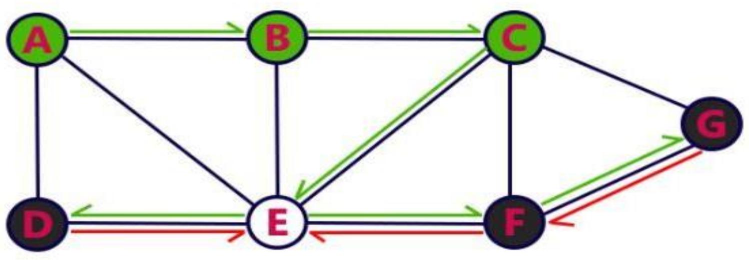
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

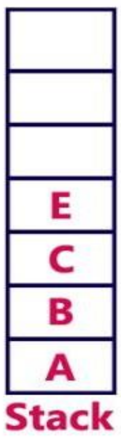




Step 10:

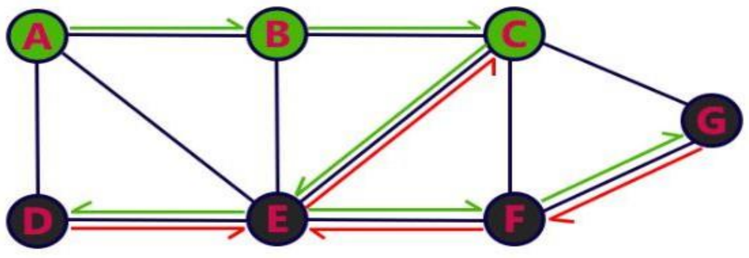
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

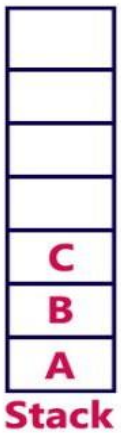




Step 11:

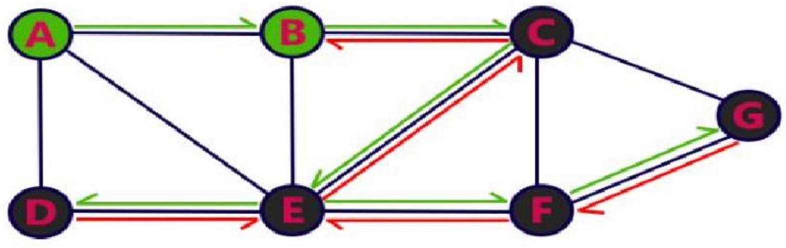
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

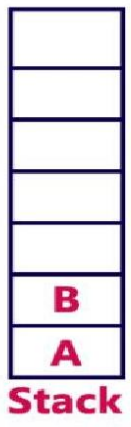




Step 12:

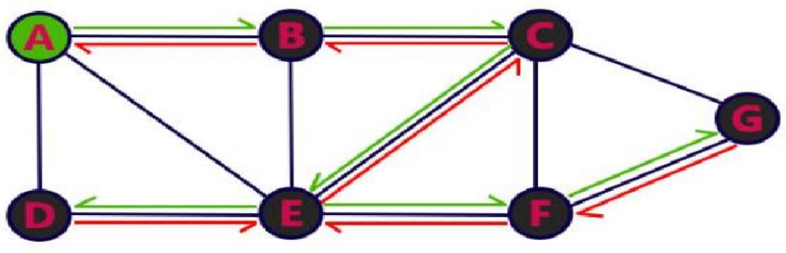
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

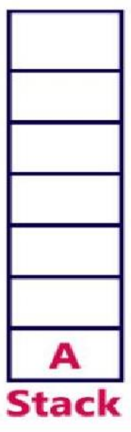


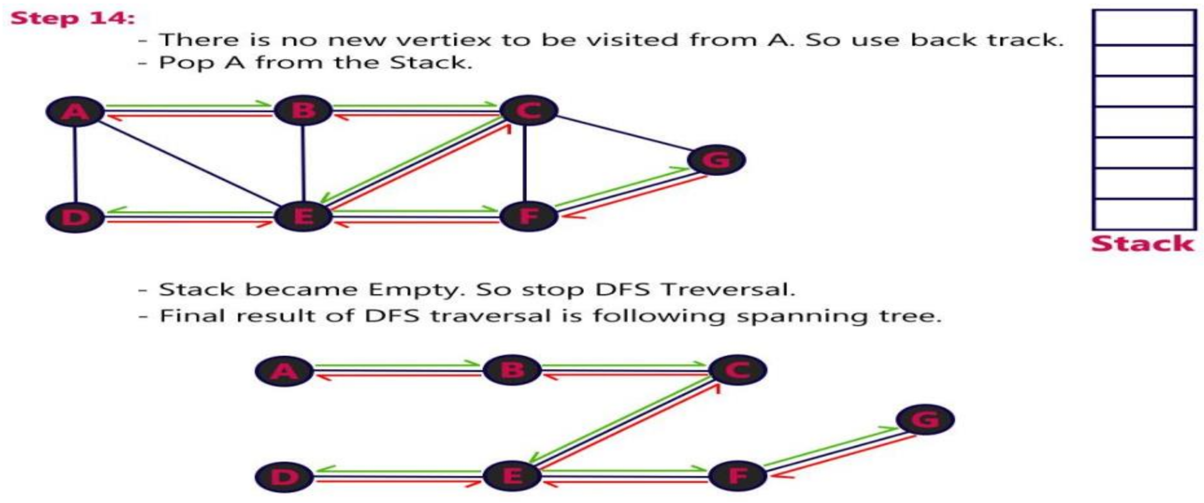


Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.







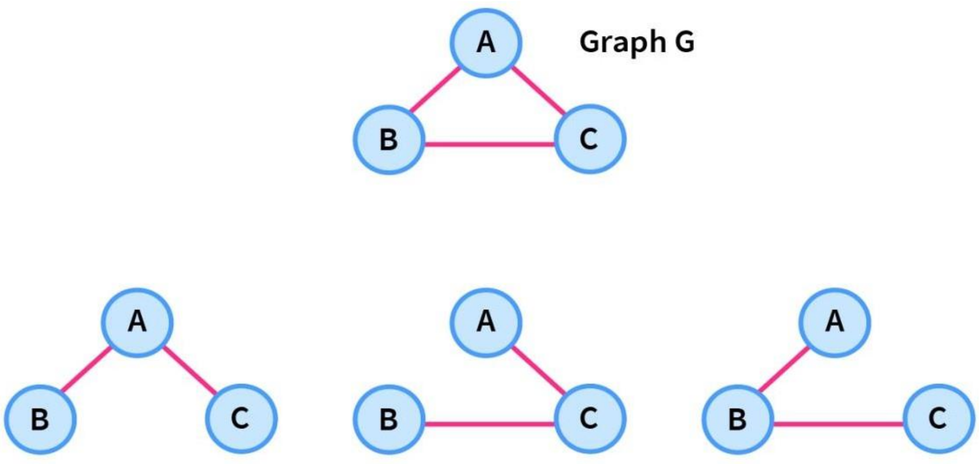
Step 14:

- 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.
- Final result of DFS traversal is following spanning tree.

SPANNING TREE: A spanning tree is a tree that connects all the vertices of a graph with the minimum possible number of edges. Thus, a spanning tree is always connected. Also, a spanning tree never contains a cycle.

Example of Spanning Tree: Let us Consider a complete graph G with 3 vertices, then the total number of spanning trees this graph can have is $3^{(3-2)}=3$ which are shown in the image below.



Spanning Trees, subgraph of G