D.N.R. COLLEGE (AUTONOMOUS): BHIMAVARAM

DEPARTMENT OF COMPUTER SCIENCE

DATA STRUCTURES USING C

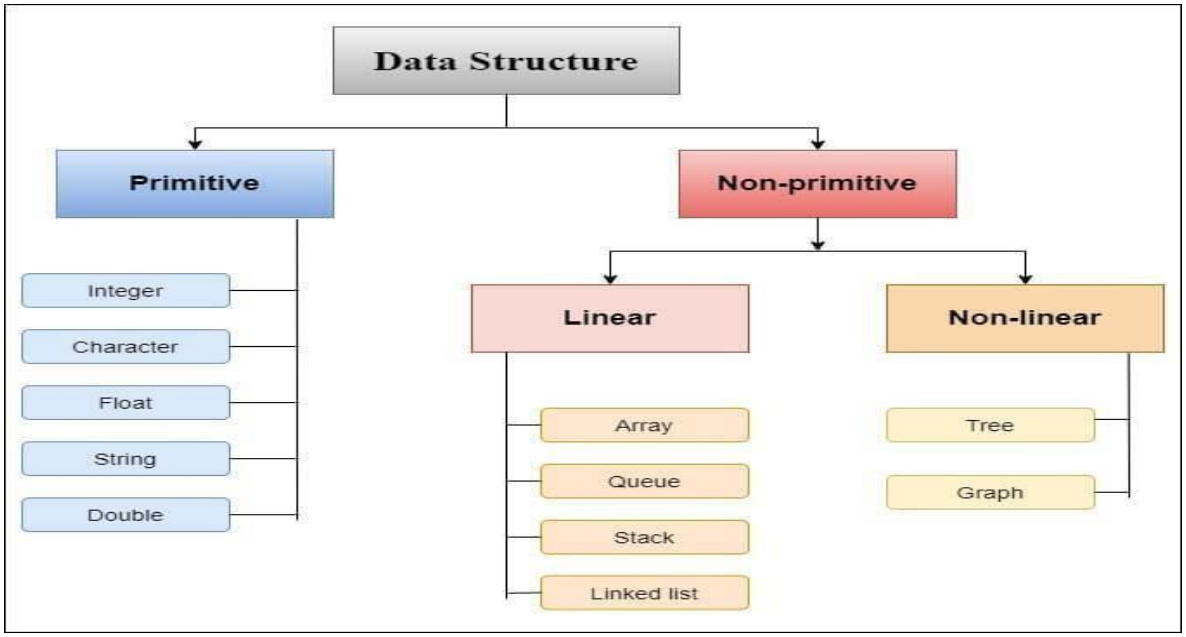
II SEMESTER PAPER-2

UNIT-1
INTRODUCTION TO DATA STRUCTURES

Data Structure:

A **data structure** is a special way of organizing and storing data in a computer so that it can be used efficiently. Array, Linked List, Stack, Queue, Tree, Graph etc. are all data structures that stores the data in a special way so that we can **access and use the data efficiently**. We have two types of data structures:

Classification of Data Structure:



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. These data 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 it for whole numbers.
- Float: It is a data type which use for storing fractionalnumbers.
- Character: It is a data type which is used for charactervalues.
- 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.

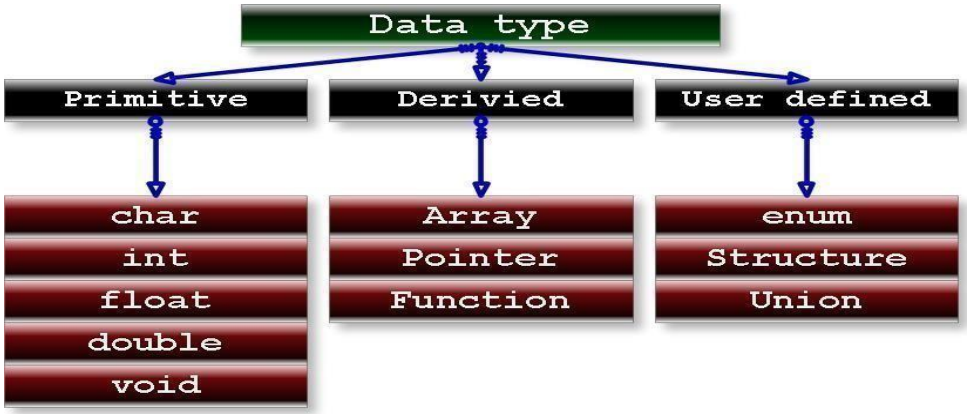
b. Non-Linear Data Structure:

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.

Data types in C:

A Data type is a set of values along with a set of rules for allowed operations. '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 data types.

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 is 1 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;

E.g. 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;

E. g: 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;

E.g. float f1, f2, f3;

d. Double Data Type :

The double data type accepts large floating value. The range of the double variable is1.7E – 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;

E.g. 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. DerivedData 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 C program has at least one function, which is main().

3. User Defined Data type:

C allows the feature called type definition which allows programmers to define their identifier that would represent an existing data type. There are three such types:

- **Enum:** Enumeration is a special data type that consists of integral constants, and each of them is assigned with a specific name. "enum" keyword is used to define the enumerated data type.
- **Structure:** It is a package of variables of different types under a single name. This is done to handle data efficiently. "struct" keyword is used to define a structure.
- **Union:** These allow storing various data types in the same memory location. Programmers can define a union with different members, but only a single member can contain a value at a given time.

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.

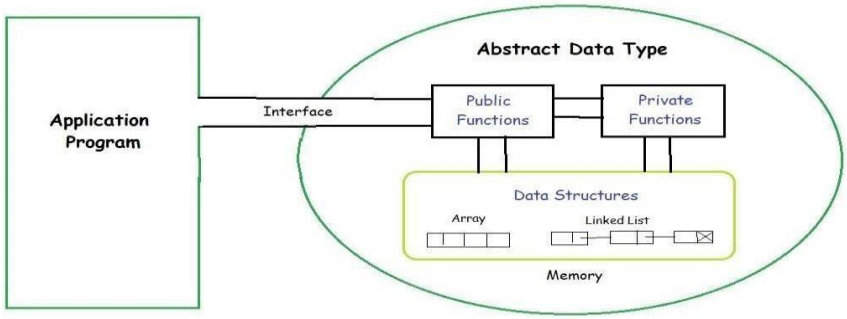
Non-Linear Data Structure:

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.

Abstract Data Type:

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value 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. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.



The user of data type does not need to know how that data type is implemented, for example, we have been using 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. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

Operations on ADT:

1. Find (key): Return a record with the given key or null if no record with the given key.

2. Insert (key, data): Insert a new record with the given key and error if the dictionary already contains a record with the given key.

3. Remove (key): Removes the record with the given key and error if there is no record with the given key.

C Programming Tips:

C is one of the most important and widely used of all programming languages. It is a powerful language that can be used not only to build general-purpose applications but also to write "low-level" programs that interact very closely with the computer hardware. Experienced C programmers have all kinds of tricks to make the most of the C language. Here is a list of the top 10 tips for both new and experienced C programmers.

1. **Function pointers**

   Sometimes it is useful to store a function in a variable. This isn't a technique that is normally used in day-to-day programming, but it can be used to increase the modularity of a program by, for example, storing the function to be used in handling an event in the event's data (or control) structure.

2. **Variable-length argument lists**

   Normally you declare a function to take a fixed number of arguments. But it is also possible to define functions capable of taking variable numbers of arguments. The standard C function printf() is a function of this sort.

3. **Testing and setting individual bits**

   Manipulating the individual bits of items such as integers is sometimes considered to be a dark 6 art used by advanced programmers. It's true that setting individual bit values can seem a rather obscure procedure. But it can be useful, and it is a technique that is well worth knowing.

4. **Short circuit operators**

   C's logical operators, && ("and") and || ("or"), let you chain together conditions when you want to take some action only when all of a set of conditions are true (&&) or when any one set of conditions is true (||). But C also provides the & and | operators.

5. **Ternary operators**

   A ternary operation is one that takes three arguments. In C the ternary operator (? can be used as a shorthand way of performing if else tests.

6. **Stacks – pushing and popping**

   A "stack" is a last-in, first-out storage system. You can use address arithmetic to add elements to a stack (pushing) or remove elements from the stack (popping). When programmers refer to "the stack", they typically mean the structure that is used by the C compiler to store local Variables declared inside a function.

7. **Copying data**

   Here are three ways of copying data. The first uses the standard C function, memcpy(), which copies n bytes from the src to the dst buffer.

8. **Testing for header inclusion**

   C uses "header" (".h") files that may contain declarations of functions and constants. A header file may be included in a C code file by importing it using its name between angle brackets when it is one of the headers supplied with your compiler (#include < string.h >) or between double-quotes when it is a header that you have written: (#include "mystring.h").

9. **Parentheses – to use or not to use?**

A competent and experienced C programmer will neither overuse nor underuse parentheses the round bracket delimiters "(" and ")". But what exactly is the correct way to use parentheses?

10. Arrays as addresses

Programmers who come to C from another language frequently get confused when C treats an array as an address and vice versa. C is correct: an array is just the base address of a block of memory, and the array notation you may have come across when learning a language, such as Java or JavaScript, is merely syntactic sugar.

Algorithm design approaches:

1. **Greedy Method:**

   In the greedy method, at each step, a decision is made to choose the local optimum, without thinking about the future consequences.

   Example: Fractional Knapsack, Activity Selection.

2. **Divide and Conquer:**

   The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer.
   Example: Merge sort, Quicksort.

3. **Dynamic Programming:**

   The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. "Dynamic" means we dynamically decide, whether to call a function or retrieve values from the table.
   Example: 0-1 Knapsack, subset-sum problem.

4. **Linear Programming:**

   In Linear Programming, there are inequalities in terms of inputs and maximizing or minimizing some linear functions of inputs.
   Example: Maximum flow of Directed Graph

5. **Reduction(Transform and Conquer):**

   In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms.

   Example: Selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called transform and conquer.

Steps to design an algorithm:

Step 1: Obtain a description of the problem. This step is much more difficult than it appears.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

Primitive Data type:

'C' supports mainly four primitive data types.

1. Character Data Type
2. Integer Data Type
3. Float Data Type
4. Double Data Type

1. 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 is 1 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;

E.g. char ch1, ch2, ch3;

2. 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;

E. g: int a, b, c;

3. 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;

E.g. float f1, f2, f3;

4. Double Data Type :

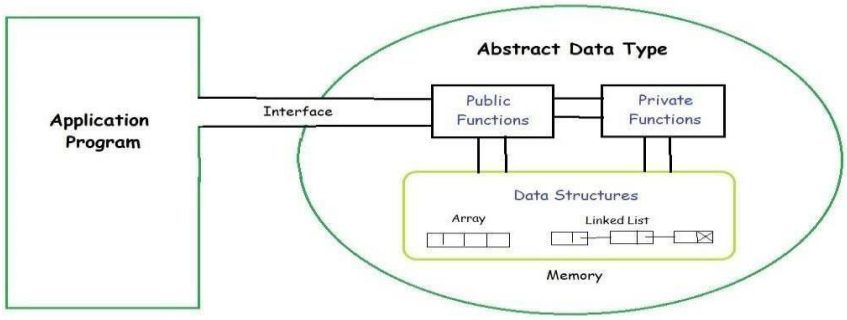
The double data type accepts large floating value. The range of the double variable is1.7E – 308 to 1.7E + 308. Double is the key word for double data type. In generally 8 bytes of memory is required to store double value.

Syntax: double list of variables;

E.g. double d, e, f;

Abstract Data Type:

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value 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. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.



The user of data type does not need to know how that data type is implemented, for example, we have been using 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. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

a) Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

```
void recursion() {
  recursion(); /* function calls itself */
}

int main() {
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function. Otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

b) Algorithm Analysis:

A complete analysis of the running time of an algorithm involves the following steps:

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modelled input.
- Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

Steps to design an algorithm

Step 1: Obtain a description of the problem. This step is much more difficult than it appears.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

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 structure are integer, character, and float.	Examples of non-primitive data structure are Array, Linked list, stack.
Primitive data structure will contain some value, i.e., it cannot be NULL.	Non-primitive data structure can consist of a NULL value.
The size depends on the type of the data structure.	In case of non-primitive data structure, size is not fixed.
It starts with a lowercase character.	It starts with an uppercase character.
Primitive data structure can be used to call the methods.	Non-primitive data structure cannot be used to call the methods.

Algorithm:

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input and produces a desired output.

For example,

An algorithm to add two numbers:

1. Take two number inputs
2. Add numbers using the + operator
3. Display the result

Qualities of Good Algorithms

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Example: Add two numbers entered by the user

Step 1: Start

Step 2: Declare variables num1,num2 and sum.

Step 3: Read values num1 and num2

Step4: Add num1 and num2 and assign the result to sum.

Step5: Display sum

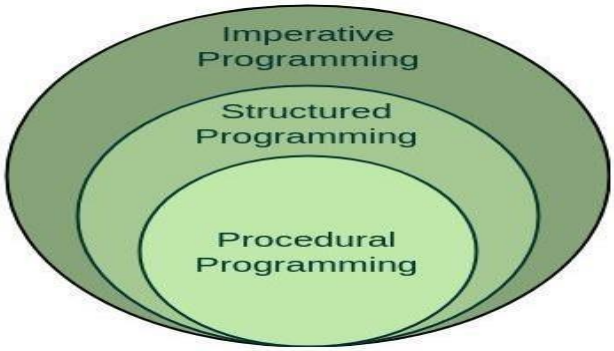
Step6: Stop

Structure Programming Approach:

Structured Programming Approach, as the word suggests, can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are:

- C
- C++
- Java
- C#

..etc.



On the contrary, in the Assembly languages like Microprocessor 8085, etc., the statements do not get executed in a structured manner. It allows jump statements like GOTO. So the program flow might be random.

The structured program mainly consists of three types of elements:

- Selection Statements
- Sequence Statements
- Iteration Statements

The structured program consists of well-structured and separated modules. But the entry and exit in a structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore a structured program is well maintained, neat and clean program. This is the reason why the Structured Programming Approach is well accepted in the programming world.

Refinement stages:

Refinement is the idea that software is developed by moving through the levels of abstraction, beginning at higher levels and, incrementally refining the software through each level of abstraction, providing more detail at each increment. At higher levels, the software is merely its design models; at lower levels there will be some code; at the lowest level the software has been completely developed.

Software engineering:

Software engineering is **a detailed study of engineering to the design, development and maintenance of software**. Software engineering was introduced to address the issues of low-quality

software projects. Problems arise when a software generally exceeds timelines, budgets, and reduced levels of quality.

Complexity:

The purpose of Complexity is to report important advances in the scientific study of complex systems. Complex systems are characterized by interactions between their components that produce new information — present in neither the initial nor boundary conditions — which limit their predictability.

Big „O"Notation:

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

UNIT-II

Array:

An array is a special type of variable used to store multiple values of same data type at a time.

(Or)

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

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

1. Single 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 : **datatype arrayName [size] ;**

Example Code: **int rollNumbers [60] ;**

The above declaration of single dimensional array reserves 60 continuous memory locations of 2 bytes each 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 initial values.

Syntax: **datatype arrayName [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.

We can also use the following general syntax to initialize a single dimensional array without specifying size and with initial values...

datatype arrayName [ ] = {value1, value2, ...} ;

The array must be initialized if it is created without specifying any size. In this case, the size of the array is decided based on the number of values initialized.

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

char studentName [] = "btechsmartclass" ;

In the above example declaration, size of the array 'marks' is 6 and the size of the array 'studentName' is 16. This is because in case of character array, compiler stores one extra character called \0 (NULL) at the end.

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 square braces. 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 [indexValue]**

Example Code: marks [2] = 99 ;

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

2. Two Dimensional Array:

The 2-D arrays are used to store data in the form of table. We also use 2-D arrays to create mathematical **matrices**.

Declaration of Two Dimensional Array

We use the following general syntax for declaring a two dimensional array...

Syntax: **datatype arrayName [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 each in 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 number of rows and coloumns with initial values.

datatype arrayName [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. Here the 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: **matrix A [0][1] = 10 ;**

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

One Dimensional Array with an example:

Single Dimensional Array (or) 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 : **datatype arrayName [size] ;**

Example Code: **int rollNumbers [60] ;**

The above declaration of single dimensional array reserves 60 continuous memory locations of 2 bytes each 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 initial values.

Syntax: **datatype arrayName [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.

We can also use the following general syntax to initialize a single dimensional array without specifying size and with initial values...

datatype arrayName [ ] = {value1, value2, ...} ;

The array must be initialized if it is created without specifying any size. In this case, the size of the array is decided based on the number of values initialized.

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

char studentName [] = "btechsmartclass" ;

In the above example declaration, size of the array 'marks' is 6 and the size of the array 'student Name' is 16. This is because in case of character array, compiler stores one extra character called \0 (NULL) at the end.

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 square braces. 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 [indexValue]**

Example Code: **marks [2] = 99 ;**

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

Two Dimensional Array with an example:

Two Dimensional Array:

The 2-D arrays are used to store data in the form of table. We also use 2-D arrays to create mathematical **matrices**.

Declaration of Two Dimensional Array

We use the following general syntax for declaring a two dimensional array...

Syntax: **datatype arrayName [rowSize] [columnSize] ;**

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:

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

1. Single Linked list:

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other.
The formal definition of a single linked list is as follows...

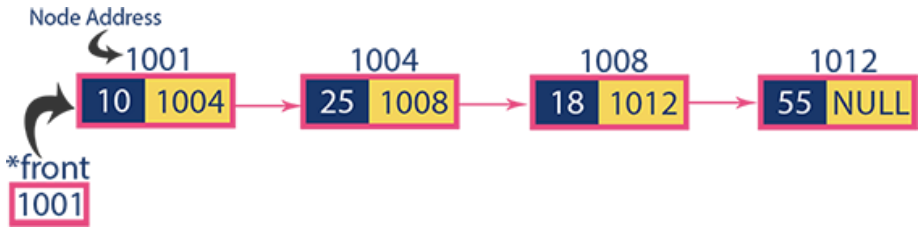
"Single linked list is a sequence of elements in which every element has link to its next element in the sequence."

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

Important Points to be Remembered

- In a single linked list, the address of the first node is always stored in a reference node known as "front" (Sometimes it is also known as "head").
- Always next part (reference part) of the last node must be NULL.

Example

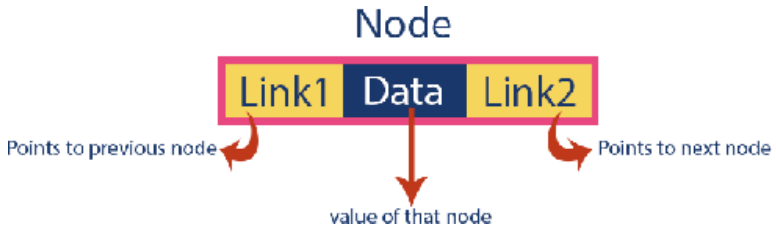


2. Double linked list:

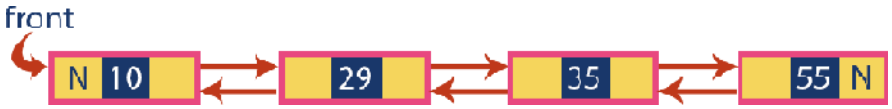
In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

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

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



Example



Important Points to be Remembered

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

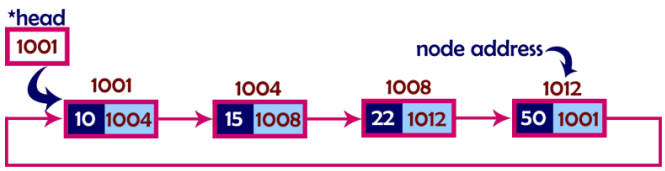
3. Circular linked list:

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

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

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

Example



Operations on Single Linked List

The following operations are performed on a Single Linked List

I. **Insertion**
II. **Deletion**
III. **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...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

1. Inserting At Beginning of the list

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

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

2. 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 **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Set **temp → next** = **newNode**.

3. Inserting At Specific location in the list (After a Node)

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

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

- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

❖ Deletion

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

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

1. Deleting from Beginning of the list

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

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → next** == **NULL**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

2. Deleting from End of the list

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

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Steps 3 - If it is Not Empty then, define** two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1 → next** == **NULL**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)
- **Step 7 -** Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

3. Deleting a Specific Node from the list

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

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3 -** If it is **Not Empty** then, 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)**).

❖ Display

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

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!!**' and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **head**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (--->) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Advantages and Disadvantages of Pointer:

Advantages:

- Pointers provide direct access to memory
- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements Pointers can be used to pass information back and forth between the calling function and called function

- Pointers allow us to perform dynamic memory allocation and deallocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees,graphs etc.
- Pointers allow us to resize the dynamically allocated memory block.
- Addresses of objects can be extracted using pointers

Disadvantages:

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Dynamic Memory Allocation:

- Allocation of memory at the time of execution (run time) is known as dynamic memory allocation.
- The functions calloc() and malloc() support allocating of dynamic memory.
- Dynamic allocation of memory space is done by using these functions when value is returned by functions and assigned to pointer variables.
- In this case, variables get allocated only if your program unit gets active.
- It uses the data structure called heap for implementing dynamic allocation.
- There is memory reusability and memory can be freed when not required.
- It is more efficient.
- In this memory allocation scheme, execution is slower than static memory allocation.
- Here memory can be released at any time during the program.

Write a C program to traverse the elements by using Two Dimensional Array.

Program:

```
#include<stdio.h>
int main()
{
int i=o,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
for(i=0;i<4;i++)
{
For(j=0;j<3;j++)
```

```
    {
    Printf("arr[%d][%d]=%\n",i,j,arr[i][j]);
    }
    }
return 0;
```

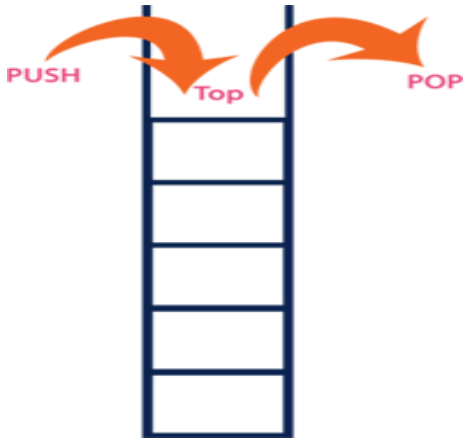
Difference between Arrays and Linked Lists.

Arrays	Linked Lists
An array is a collection of elements of a similar data type.	Linked List is an ordered collection of elementsof the same type in which each element is connected to the next using pointers.
Array elements can be accessed randomly using the array index.	Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.
Data elements are stored in contiguous locations in memory.	New elements can be stored anywhere and a reference is created for the new element using pointers.
Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	Insertion and Deletion operations are fast and easy in a linked list.
Memory is allocated during the compile time (Static memory allocation).	Memory is allocated during the run-time (Dynamic memory allocation).
Size of the array must be specified at the time of array declaration/initialization.	Size of a Linked list grows/shrinks as and when new elements are inserted/deleted.

UNIT-III

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 fromthe top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



Operations on a Stack

The following operations are performed on the stack...

1. **Push (To insert an element on to the stack)**
2. **Pop (To delete an element from the stack)**
3. **Display (To display elements of the stack)**

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 A, B, C, D, E, it represents the stack of five elements. In figure (a), we want to push 'A' element on the stack then the top becomes zero (top=0), similarly the top=1 when 'B' element is pushed, top=2 when the 'C' element is pushed, top=3 when the 'D' element is pushed, and top=4 when the 'E' 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 'E' element has to be deleted first. The push operation is shown in the below figure.

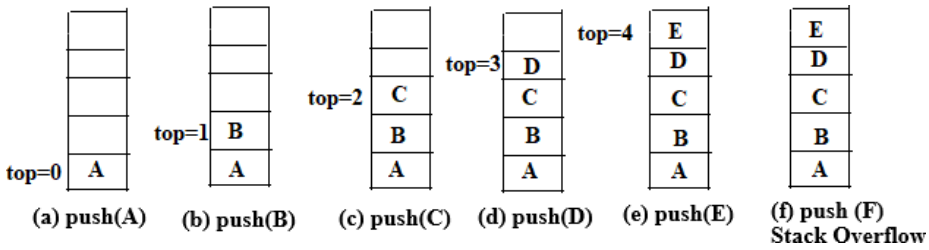


Fig: Push Operation

2. <u>Pop (To delete an element from the stack):</u>

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 'E' element is deleted next 'D' element…..' A'. 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.

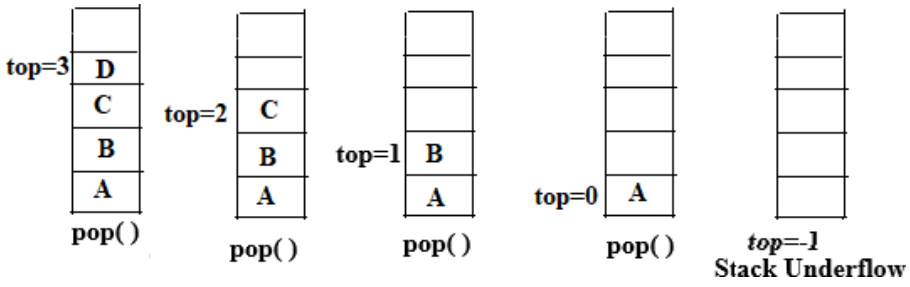
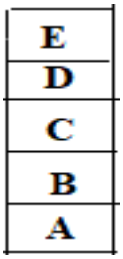


Fig: POP Operation

So this is the explanation of how the elements are inserted and deleted in the stack by using push and pop operation.

3. **Display :**

Display() is used to display all the elements in the stack

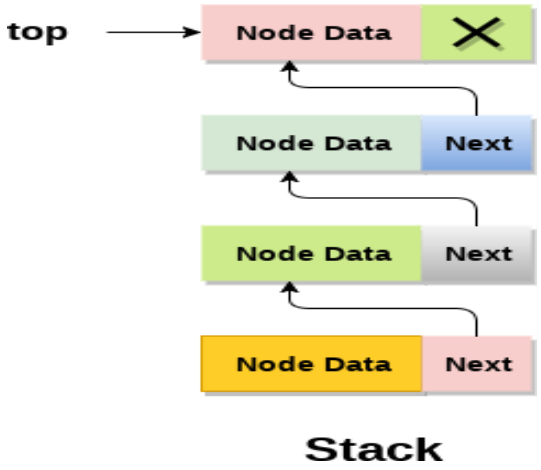


It displays elements as follows **E, D, C, B, A.**

Stack using Linked List with an example:

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.

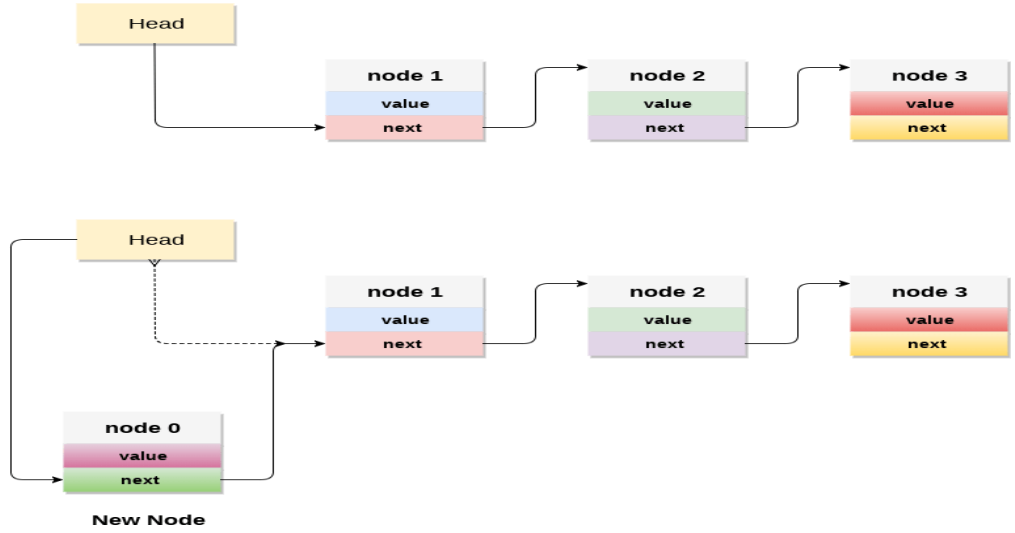


Stack Operations using Linked List:

1. Push (value) - Inserting an element into the Stack

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.



2. Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

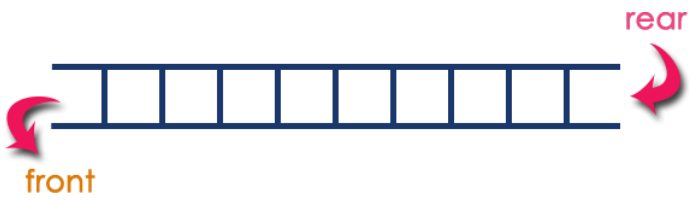
3. Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Queue :

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



Example

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

Types of Queues:

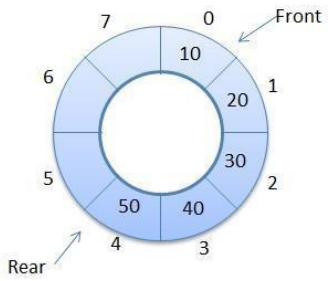
1. Simple Queue

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

All nodes are connected to each other in a sequential manner. The pointer of the first node points to the value of the second and so on.

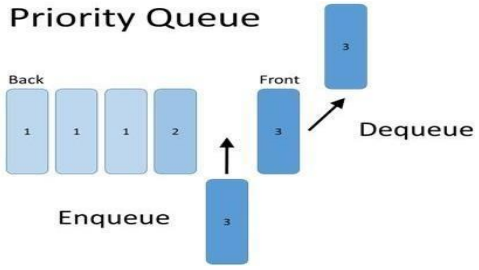
The first node has no pointer pointing towards it whereas the last node has no pointer pointing out from it.

2. Circular Queue



Unlike the simple queues, in a circular queue each node is connected to the next node in sequence but the last node's pointer is also connected to the first node's address. Hence, the last node and the first node also gets connected making a circular link overall.

3. Priority Queue



Priority queue makes data retrieval possible only through a pre-determined priority number assigned to the data items.

While the deletion is performed in accordance to priority number (the data item with highest priority is removed first), insertion is performed only in the order.

4. Doubly Ended Queue (Dequeue)

The doubly ended queue or dequeue allows the insert and delete operations from both ends (front and rear) of the queue.

Queues are an important concept of the data structures and understanding their types is very necessary for working appropriately with them.

Implementation of Queues:

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

1. **Using Array**
2. **Using Linked List**

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

1. Queue Data structure Using Array

A Queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables '**front**' and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Array

- **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.

- **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.

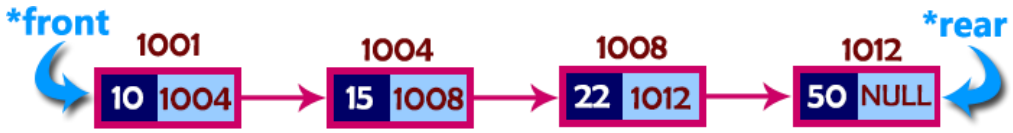
display() - Displays the elements of a Queue

2. Queue Using Linked List

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

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

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



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

Operations

Following are basic operations of Queue:

Main Queue Operations:

1) **EnQueue()**: Inserts an element at the rear of the Queue.

2)**DeQueue()**: Remove and return the front element of the Queue.

Applications of Stack:

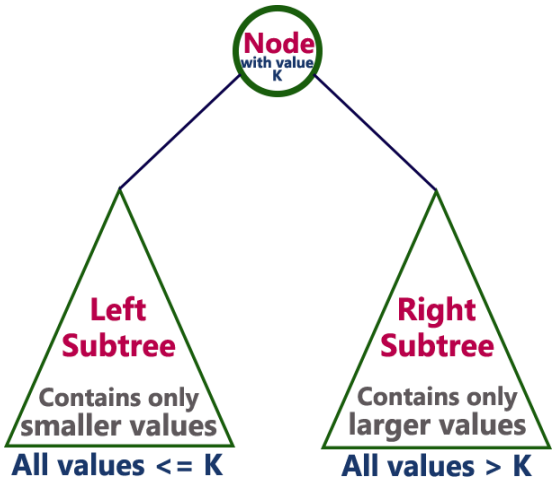
1. Expression Evaluation and Conversion
2. Backtracking
3. Parenthesis Checking
4. Function Call
5. String Reversal
6. Syntax Parsing
7. Memory Management

UNIT-IV

Construct a BST:

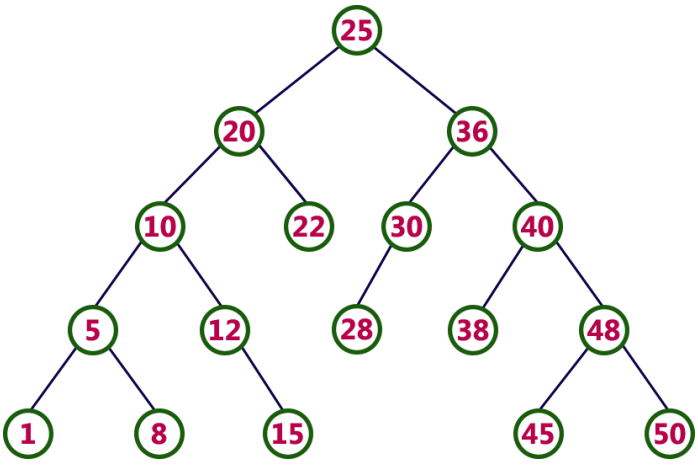
Binary Search Tree is a binary tree in which every node contains only smaller values in its left sub tree and only larger values in its right sub tree.

In a binary search tree, all the nodes in the left sub tree of any node contains smaller values and all the nodes in the right sub tree of any node contains larger values as shown in the following figure...



Example

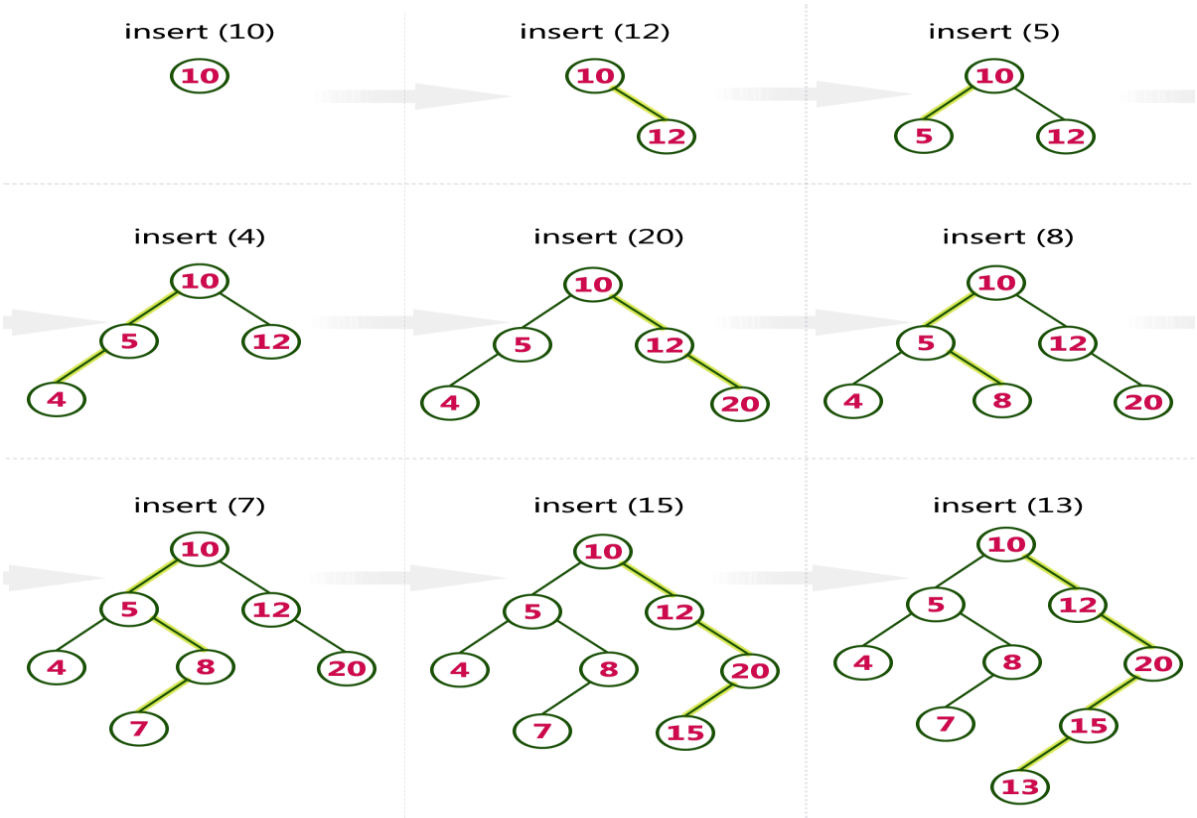
The following tree is a Binary Search Tree. In this tree, left sub tree of every node contains nodes with smaller values and right sub tree of every node contains larger values.



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...



Different operations on a Binary Search Tree:

Operations on binary search tree:

The following operations are performed on binary search tree

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

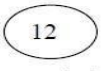
Constructing a binary search tree:

1. Take the first element as the root of the tree.
2. Take the next element and compare with root node. If it is less than the root node then it is added to left sub tree.
3. If the element is greater than the root node, then it is added to right sub tree.
4. Repeat the steps 2&3 until the last element.

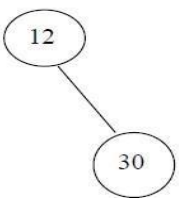
Example: 12, 30, 11, 90, 75, 9, 2

Insertion:

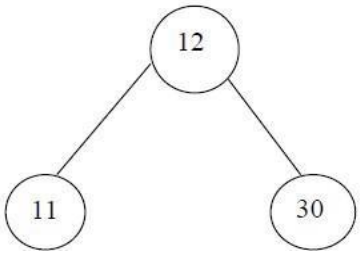
- Insert 12 into the BST, if tree is not available then create root node and place 12 into it.



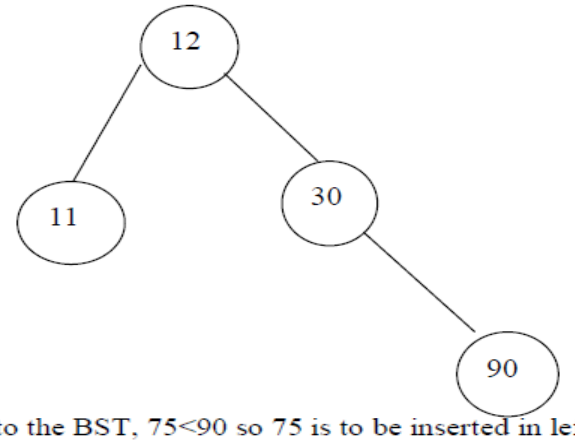
- Insert 30 into the BST, 30>12 so, 30 is inserted in the right sub tree of 12



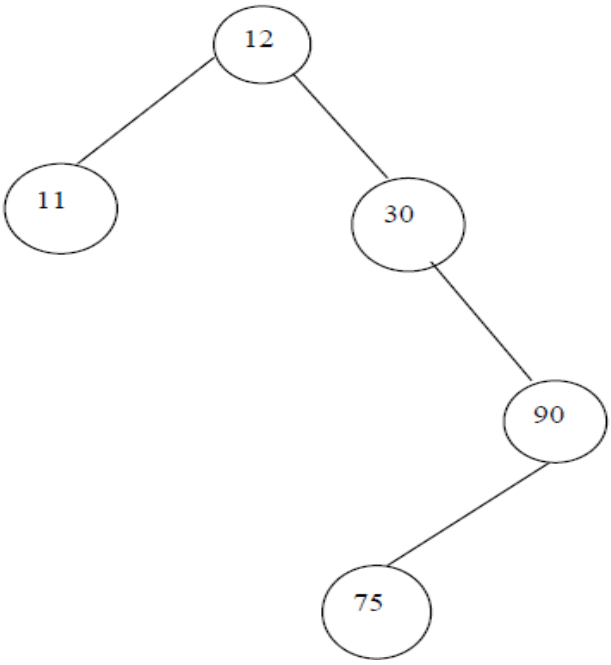
- Insert 11 into the BST, 11<12 so 11 is inserted in the left sub tree 12.



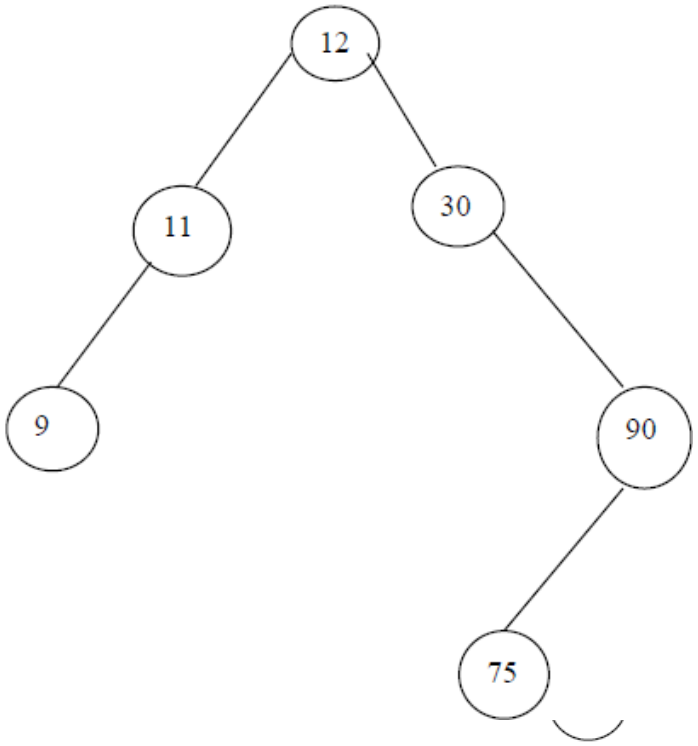
- Insert 90 into the BST, so 90 is to be inserted right sub tree of 30.



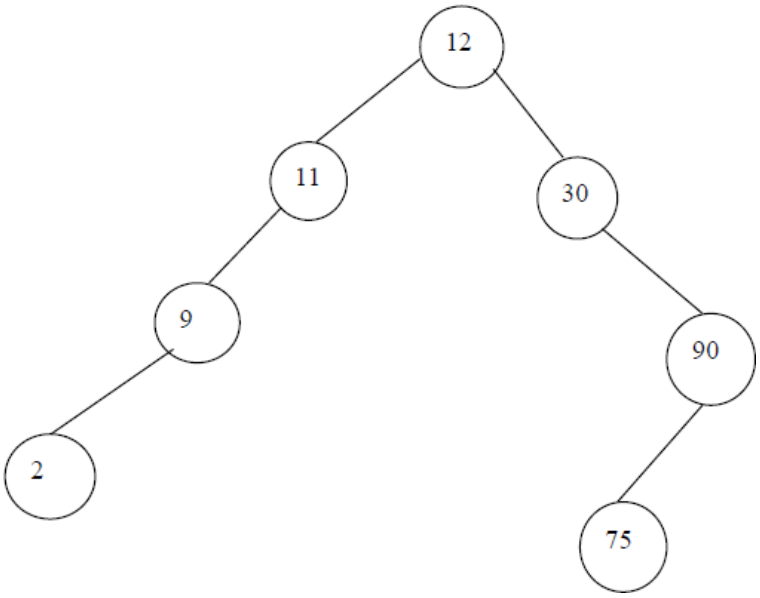
- Insert 75 into the BST, 75<90 so 75 is to be inserted in left sub tree of 90.



- Insert 9 into the BST, 9<11. So 9 is to be inserted left sub tree of 11

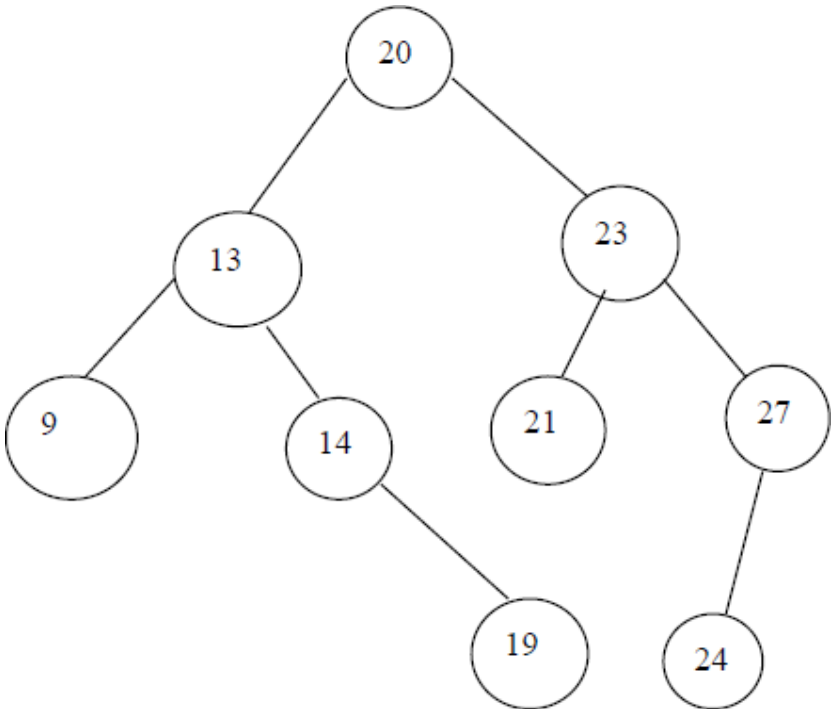


- Insert 2 into the BST, 2<9 So 2 is to be inserted left sub tree of 9.

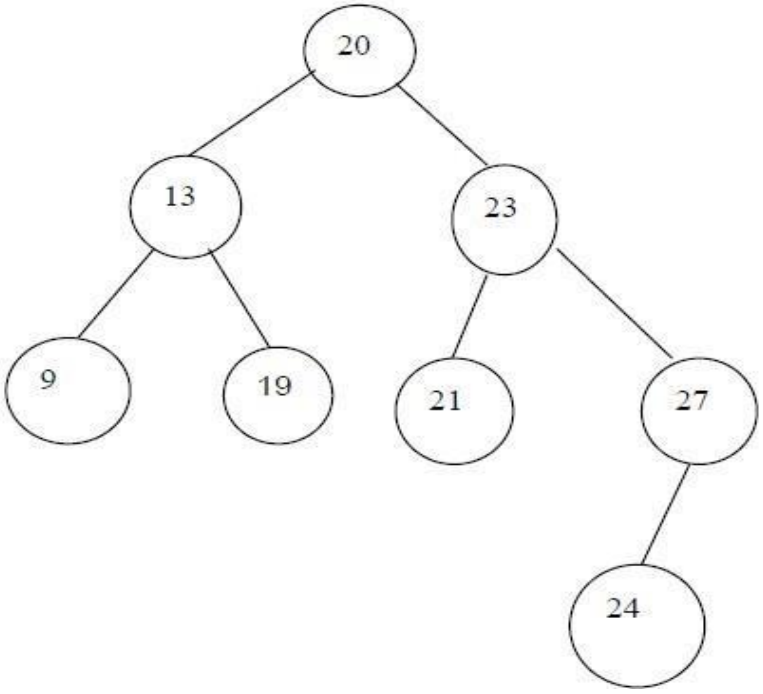


Deletion:

- Deleting a node from binary search tree.
- A node can be deleted from BST, in 3 positions.
- A node is to be deleted which has no children can be done by placing null inits parent node

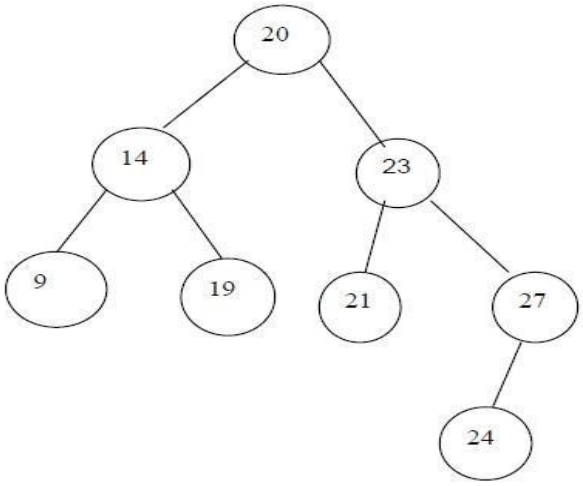


- A node with one child is to be deleted in this case, parent node is replacing by a child.



Forest: Forest is an ordered set of ordered trees

- A node with two children is to be deleted. For example delete 13 from the above BST.



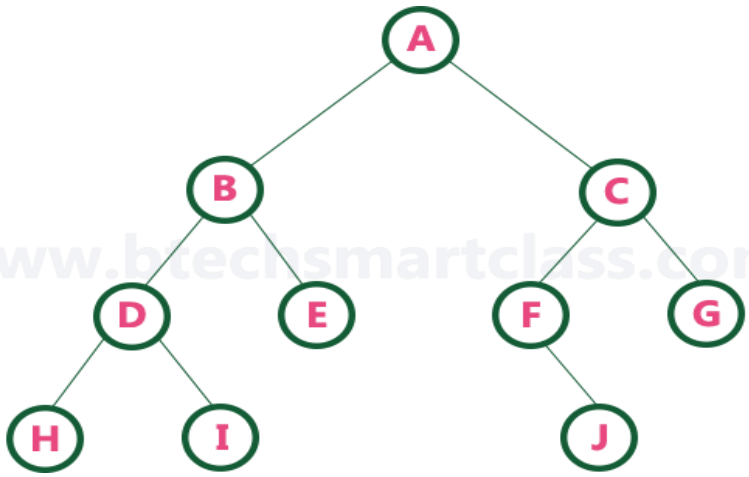
Binary Tree:

In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as a left child and the other is known as right child.

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

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

Example:



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

1. Strictly Binary Tree
2. Complete Binary Tree
3. Extended Binary Tree

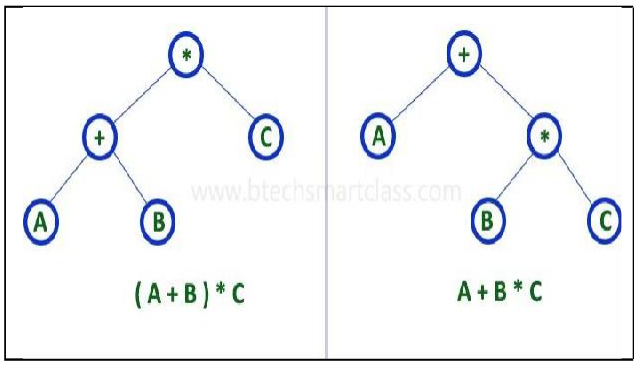
1. Strictly Binary Tree:

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

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. Strictly binary tree is also called as Full Binary Tree **or** Proper Binary Tree **or** 2-Tree

A strictly Binary Tree can be defined as follows...

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

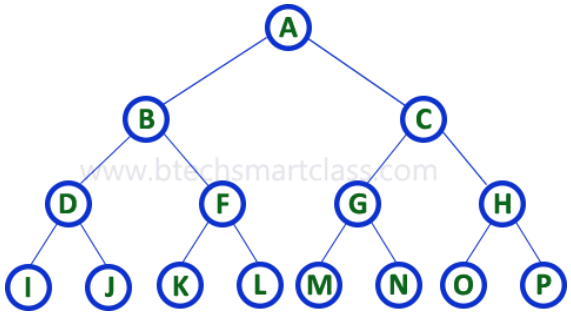


2 .Complete Binary Tree:

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

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes. Complete binary tree is also called as **Perfect Binary Tree**

Example

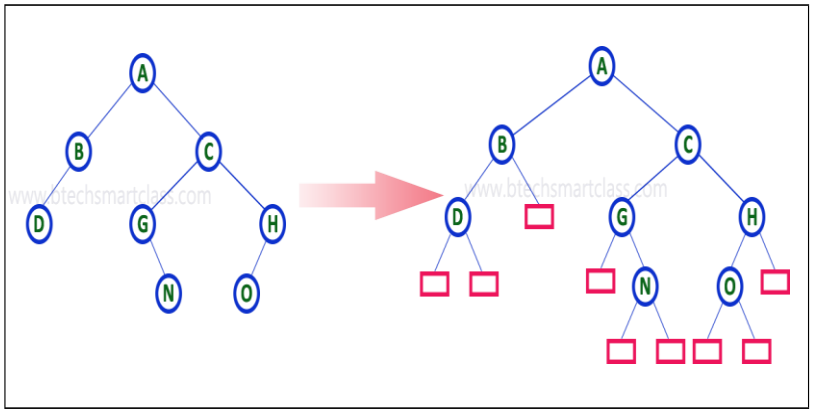


3. Extended Binary Tree:

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

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required. In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

Example:



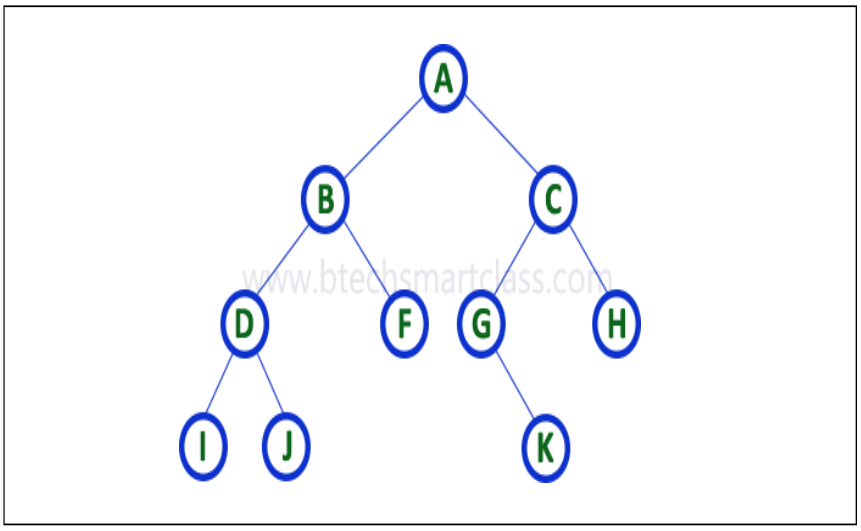
Binary Tree Traversals.

Binary Tree Traversals: Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

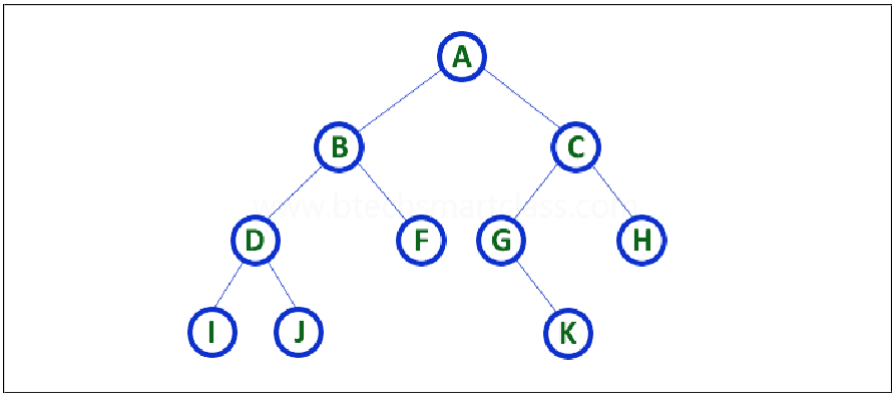
1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



1. In - Order Traversal (left Child - root - right Child)

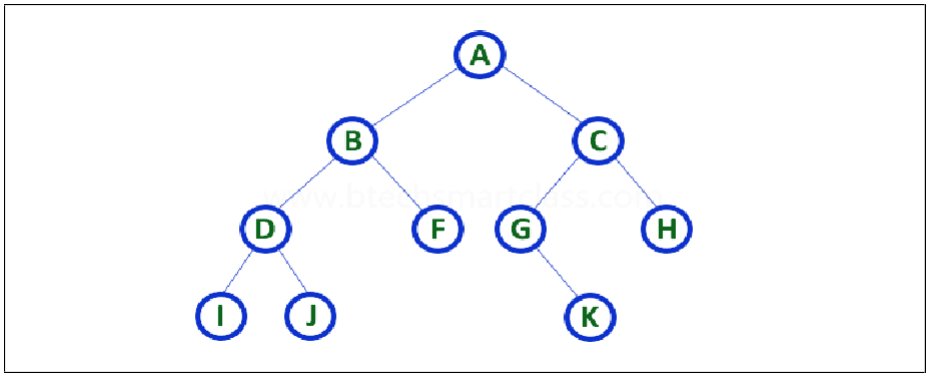
In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all sub trees in the tree. This is performed recursively for all nodes in the tree. In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left sub tree. so we try to visit its (B's) left child 'D' and again D is a root for sub tree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**. With this we have completed the left part of node B. Then visit **'B'** and next B's right child **'F'** is visited. With this we have completed left part of node A. Then visit root node **'A'**. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit **'G'** and then visit G's right child K. With this we have completed the left part of node C. Then visit root node **'C'** and next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.



That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H **using In-Order Traversal.**

2. Pre - Order Traversal (root - leftChild - rightChild)

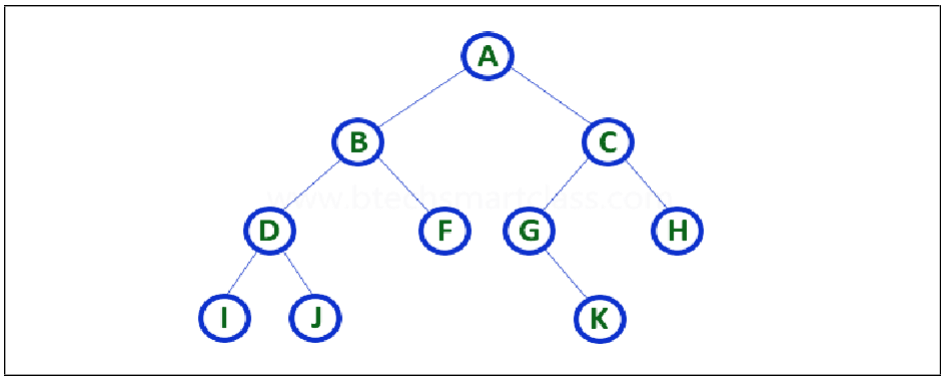
In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all sub trees in the tree. In the above example of binary tree, first we visit root node **'A'** then visit its left child **'B'** which is a root for D and F. So we visit B's left child **'D'** and again D is a root for I and J. So we visit D's left child **'I'** which is the leftmost child. So next we go for visiting D's right child **'J'**. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child **'F'**. With this we have completed root and left parts of node A. So we go for A's right child **'C'** which is a root node for G and H. After visiting C, we go for its left child **'G'** which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child **'K'**. With this, we have completed node C's root and left parts. Next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.



That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H **using Pre-Order Traversal.**

3. Post - Order Traversal (leftChild - rightChild - root):

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.



Here we have visited in the order of I - J - D - F - B - K - G - H - C - A **using Post-OrderTraversal**

Full Binary Tree ,Prefect Binary Tree:

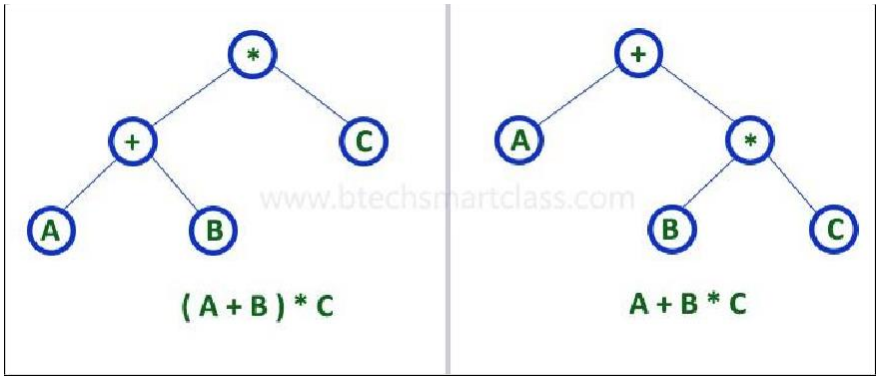
a) Full Binary Tree:

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

In a binary tree, every node can have a maximum of two children. But in full binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. Full binary tree is also called as Full Binary Tree**.**

A Full Binary Tree can be defined as follows...

Example: Full binary tree data structure is used to represent mathematical expressions

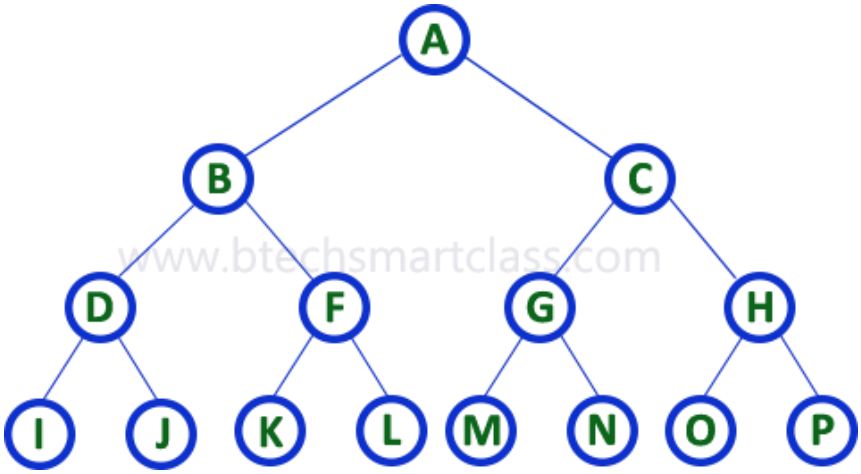


b) **Perfect Binary Tree:**

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

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in Perfect binary tree all the nodes must have exactly two children and at every level of Perfect binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

Example:



Applications of Binary Tree:

The following are the applications of binary trees:

Binary Search Tree - Used in many search applications that constantly show and hide data, such as data. For example, map and set objects in many libraries.

Binary Space Partition - Used in almost any 3D video game to determine which objects need to be rendered.

Binary Tries - Used in almost every high-bandwidth router to store router tables.

Syntax Tree - Constructed by compilers and (implicit) calculators to parse expressions.

Hash Trees - Used in P2P programs and special image signatures that require a hash to be validated, but the entire file is not available.

Heaps - Used to implement efficient priority queues and also used in heap sort.

Treap - Randomized data structure for wireless networks and memory allocation.

T-Tree - Although most databases use a form of B-tree to store data on the drive, databases that store all (most) data often use T-trees.

properties of a Tree:

Some basic properties of a binary tree:

1. A binary tree can have a maximum of nodes at level if the level of the root is zero.
2. When each node of a binary tree has one or two children, the number of leaf nodes (nodes with no children) is one more than the number of nodes that have two children.
3. There exists a maximum of nodes in a binary tree if its height is , and the height of a leaf node is one.
4. If there exist leaf nodes in a binary tree, then it has at least levels.
5. A binary tree of nodes has minimum number of levels or minimum height.
6. The minimum and the maximum height of a binary tree having nodes are and, respectively.
7. A binary tree of nodes has null references.

UNIT-V

Merge sort:

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sub lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list.

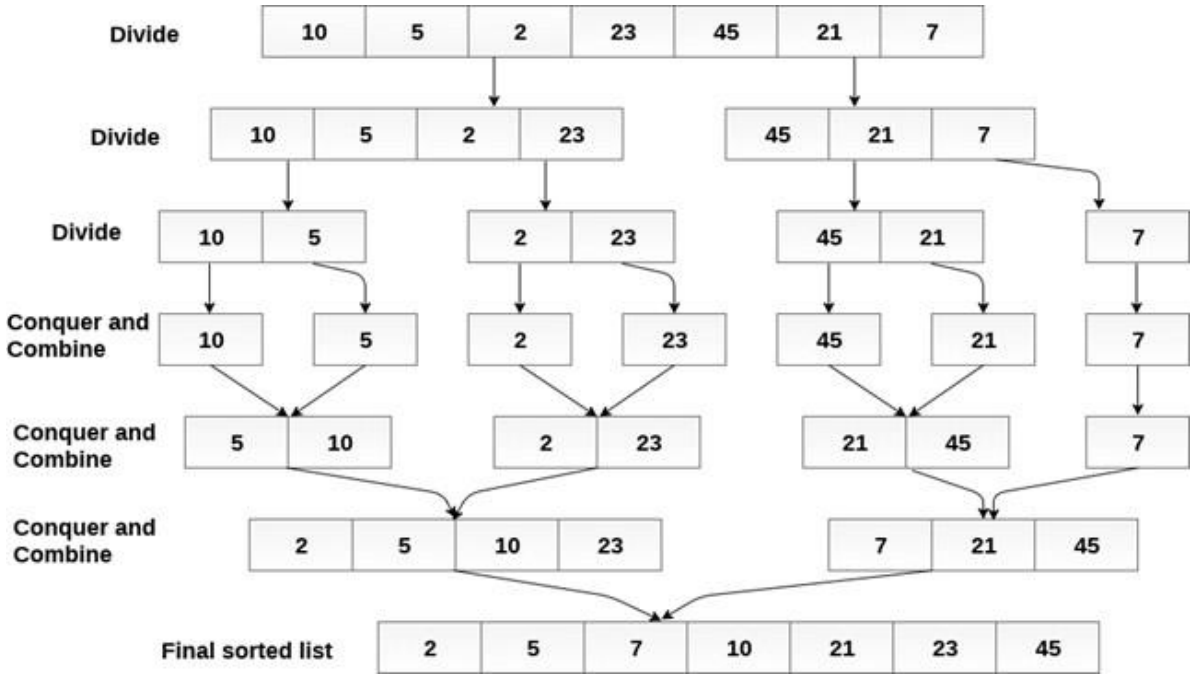
Algorithm for Merge Sort:

Step 1: Find the middle index of the array.
Middle = 1 + (last – first)/2
Step 2: Divide the array from the middle.
Step 3: Call merge sort for the first half of the array
Merge Sort(array, first, middle)
Step 4: Call merge sort for the second half of the array.
Merge Sort(array, middle+1, last)
Step 5: Merge the two sorted halves into a single so rted array.

A merge sort works as follows:
Top-down Merge Sort Implementation:

The top-down merge sort approach is the methodology which uses recursion mechanism. It starts at the Top and proceeds downwards, with each recursive turn asking the same question such as "What is required to be done to sort the array?" and having the answer as, "split the array into two, make a recursive call, and merge the results.", until one gets to the bottom of the array-tree.

Example: Let us consider an example to understand the approach better.



C program on Binary Search:

```
#include <stdio.h>
int main()
{
int i,low,high,mid,n,key,a[20];
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);
return 0;
}
```

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
```

C program on Insertion sort:

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int n, i, j, temp, a[35];
   printf("Enter number of elements\n");
   scanf("%d", &n);
   printf("Enter %d integers\n", n);
   for (i = 0; i < n; i++)
   {
      scanf("%d", &a[i]);
   }
   for (i = 1 ; i <= n - 1; i++)
   {
        j = i;
         while ( j > 0 && a[j-1] > a[j])
         {
            temp    = a[j];
            a[j]   = a[j-1];
            a[j-1] = temp;
            j--;
         }
   }

      printf("Sorted list in ascending order:\n");
      for (i = 0; i <= n - 1; i++)
      {
         printf("%d\n", a[i]);
      }
      return 0;
}
```

OUTPUT:

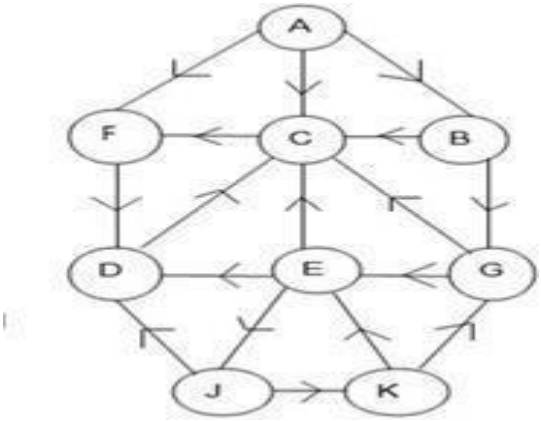
```
Enter number of elements
5
Enter 5 integers
40
10
50
70
30
Sorted list in ascending order:
10
30
40
50
70
```

Breath First Search (BFS) in Graphs.

BFS:

- Defined the size (i.e., total number of vertices in a graph)
- Prepare adjacent list, for all vertices in a graph.
- Select any vertex as a starting point for traversal, visit that vertex and insert into queue.
- Visit all adjacent vertices of the vertex which is at front of the queue, which is not visited and insert them into queue.
- When there is no new vertex, to visit from the vertex and delete that vertex from the queue.
- Repeat steps 4&5 until queue become empty.
- Finally produce a spanning tree by remaining unused edges from the graph.

Example for BFS:

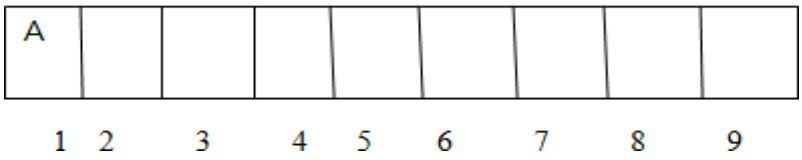


Adjacent List:

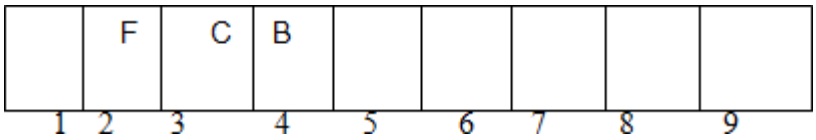
A	F, C, B
B	G, C
C	F
D	C
E	D, C, J
F	D
G	C, E
J	D, K
K	E, G

Step 1: first we prepare, adjacent list for the given graph.

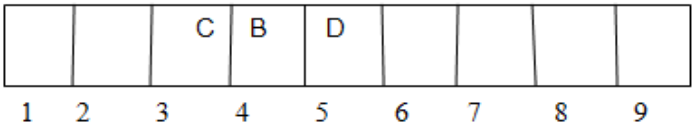
Step 2: Select the vertex A. As starting point and insert A into the Queue.



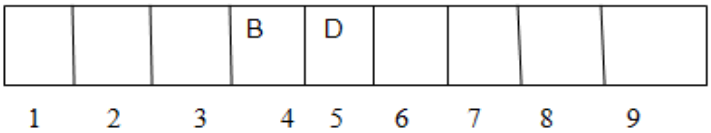
Step 3: Visit and insert all adjacent vertices of A, which not visited and delete 'A'.



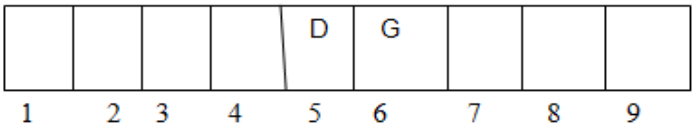
Step 4: visit and insert all adjacent vertices of F, which are not visited and delete 'F'



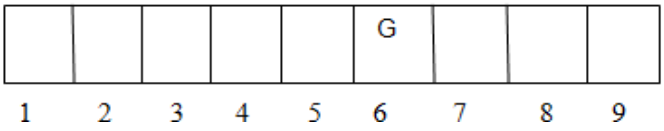
Step 5: Visit and insert all vertices of 'C' which are not visited and delete 'C'.



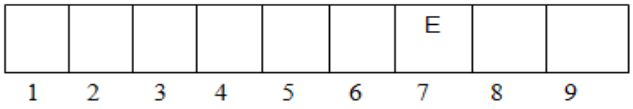
Step 6: Visit and insert all adjacent vertices of 'B' which are not visited and delete 'B'.



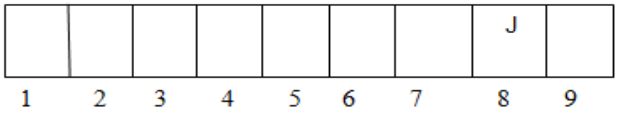
Step 7: Visit and insert all adjacent vertices of 'D' which are not visited and delete 'D'.



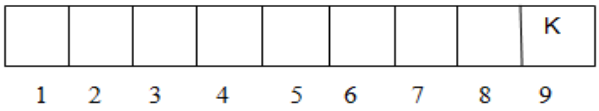
Step 8: Visit and insert all adjacent vertices of 'G' which are not visited and delete 'G'.



Step 9: Visit and insert all adjacent vertices of 'E' which are not visited and delete 'E'.



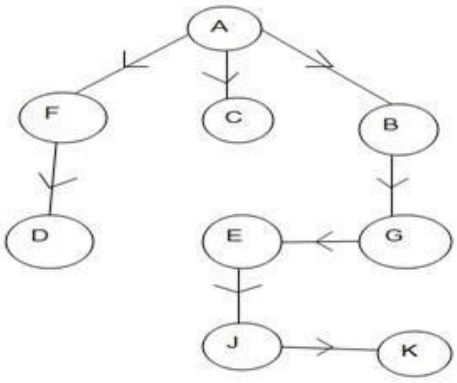
Step 10: Visit and insert all adjacent vertices of 'J' which are not visited and delete.



Step 11: Visit and insert all adjacent vertices of 'K' which are not visited and delete k.

The neighbours of 'K' are E, G which are already visited.
To produce the spanning tree we back track the edges as

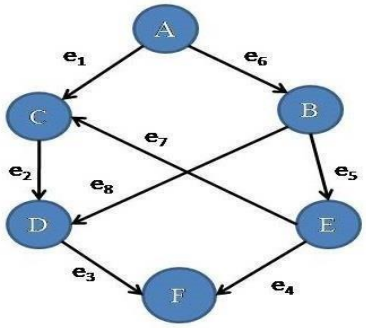
K ← J ← E ← G ← B ← A



Graph and different types of Graphs:

Graph: A graph is a ordered pair of two sets i.e., G=(V,E)

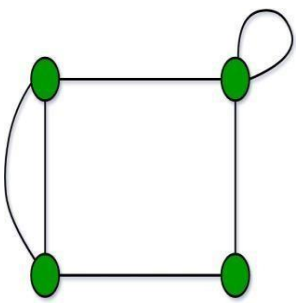
- The elements of 'v' are called vertices, where
  V={V1,V2,V3,...Vn} and
- The elements of 'E' are called edges, i.e,
  E={E1,E2,E3,... En}each edge is identified with pair of distinct vertices
- A graph is a non-linear data structure



Multi Graph: If more than one edge joining a pair of vertices, is called multi-graph

In the above example C1, C2 is called the multi graph.

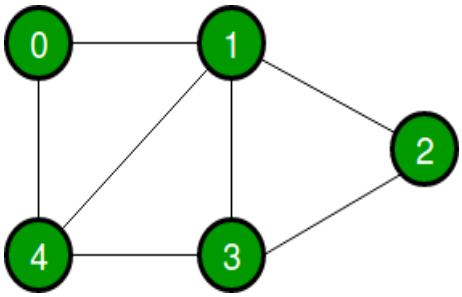
Loop: If an edge joining a vertex to itself it is called a Loop.



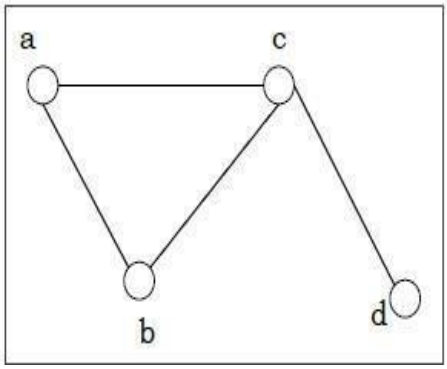
Connected Graph: A Graph is said to be connected if their path from one vertex to another vertex i.e. we can travel from any one vertex to another vertex

(or)

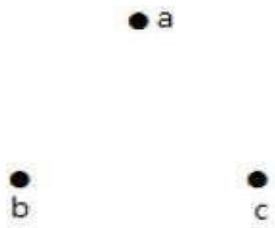
A graph is said to be connected if there is a path between every pair of vertices



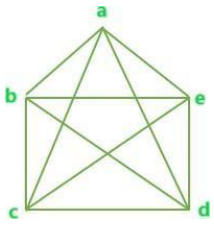
Non connected /Disconnected Graph: If there is nopath to travel from one vertex to another vertex that graph is called non-connected (or) disconnected graph



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

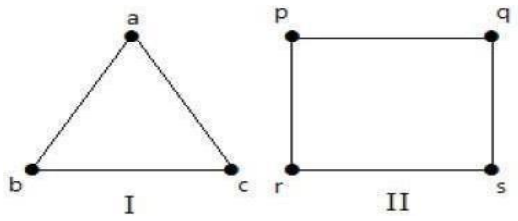


Complete graph: A graph in which every pair of vertices are adjacent then it is called complete graph



Regular Graph:

A graph in which degree of all vertices is equal then it is called regular graph



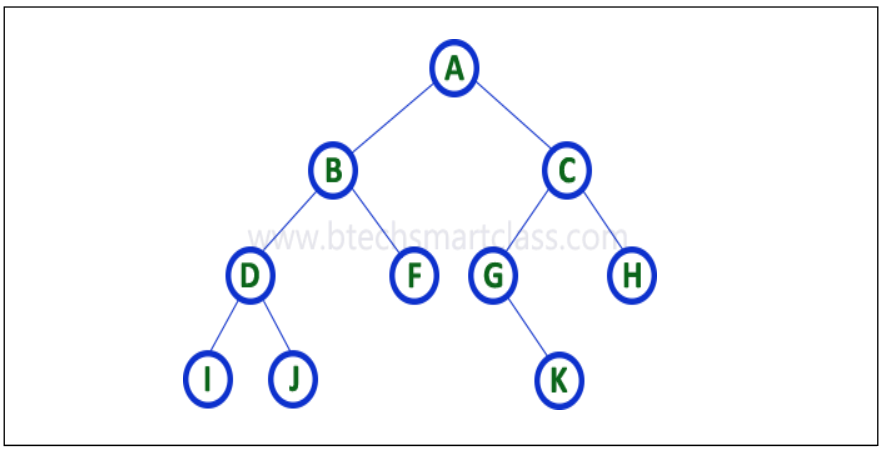
Tree Traverse Techniques:

Binary Tree Traversals: Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

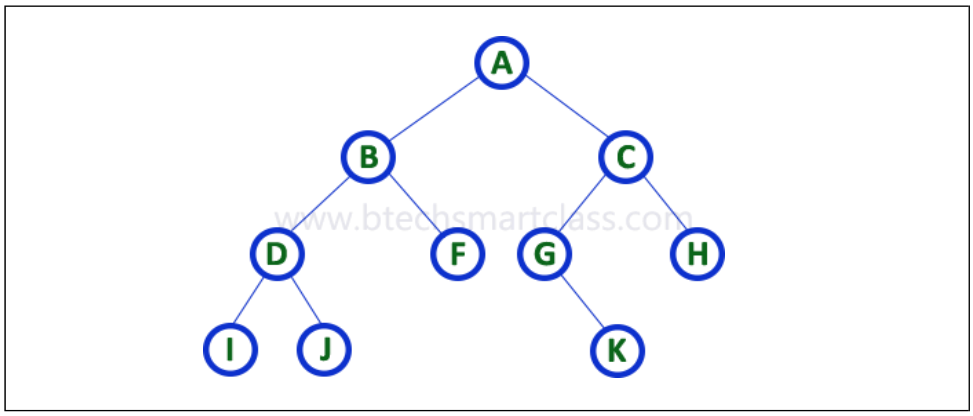
Consider the following binary tree...



1. In - Order Traversal (left Child - root - right Child)

In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all sub trees in the tree. This is performed recursively for all nodes in the tree. In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left sub tree. so we try to visit its (B's) left child 'D' and again D is a root for sub tree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**. With this we have completed the left part of node B. Then visit **'B'** and next B's right child **'F'** is visited. With this we have completed left part of node A. Then visit root node **'A'**. With this we have completed left and root parts of node A. Then we go for the

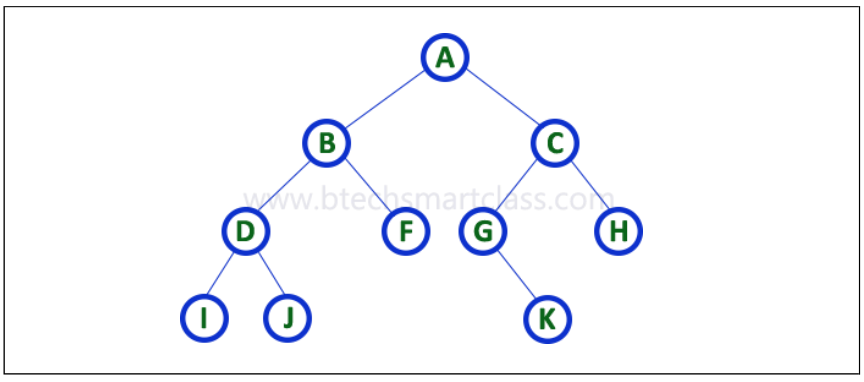
right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit **'G'** and then visit G's right child K. With this we have completed the left part of node C. Then visit root node **'C'** and next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.



That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

2. Pre - Order Traversal (root - leftChild - rightChild)

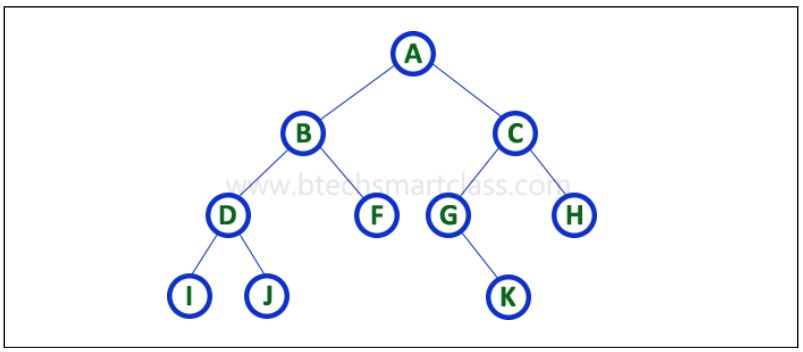
In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all sub trees in the tree. In the above example of binary tree, first we visit root node **'A'** then visit its left child **'B'** which is a root for D and F. So we visit B's left child **'D'** and again D is a root for I and J. So we visit D's left child **'I'** which is the leftmost child. So next we go for visiting D's right child **'J'**. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child **'F'**. With this we have completed root and left parts of node A. So we go for A's right child **'C'** which is a root node for G and H. After visiting C, we go for its left child **'G'** which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child **'K'**. With this, we have completed node C's root and left parts. Next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.



That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

3. Post - Order Traversal (leftChild - rightChild - root):

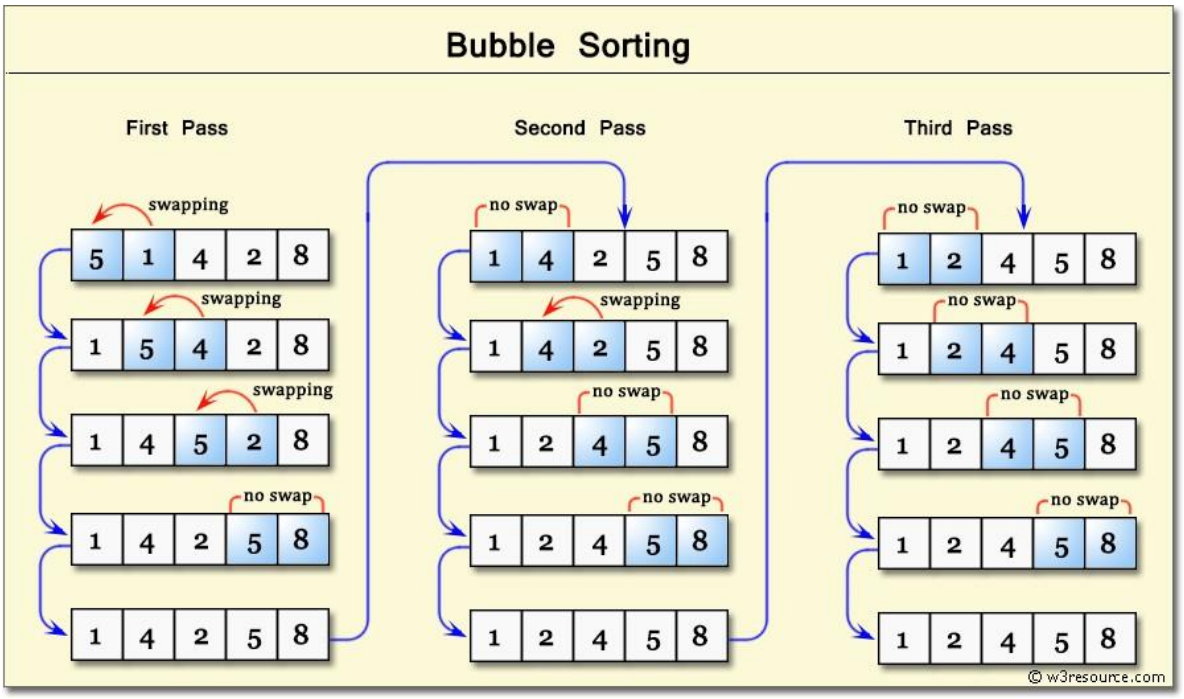
In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.



Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Bubble sort with Example:

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with 'n' number of elements. Bubble Sort compares all the element one byone and sort them based on their values.



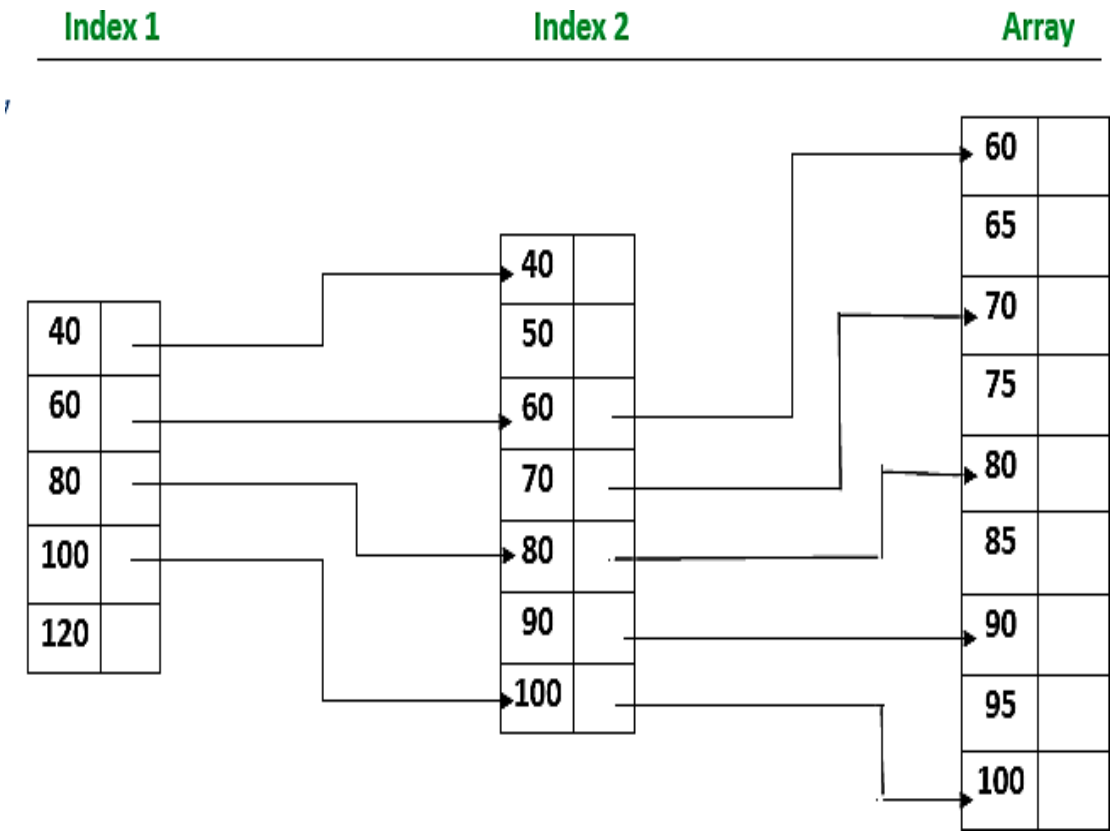
Indexed Sequential Search with Diagram:

In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time because it is located in a specified group.

Characteristics of Indexed Sequential Search:

- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array.

Explanation by diagram "Indexed Sequential Search":



Graph and its representation:

A graph representation is a technique to store graph into the memory of computer.

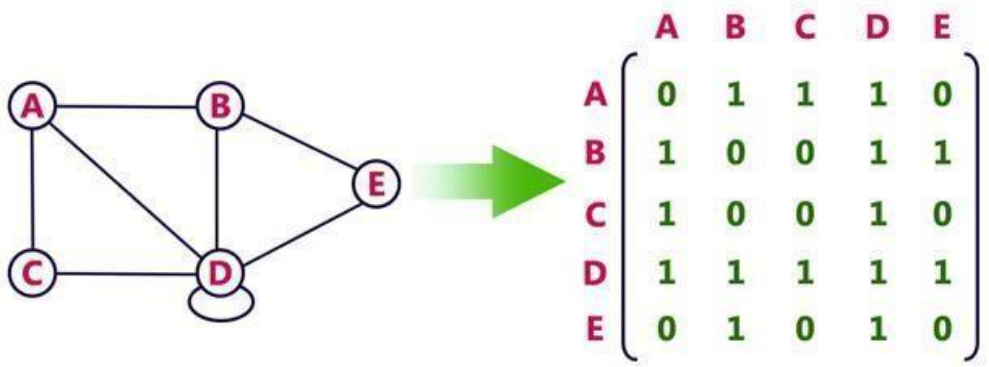
To represent a graph, we just need the set of vertices and for each vertex the neighbours of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

Example:

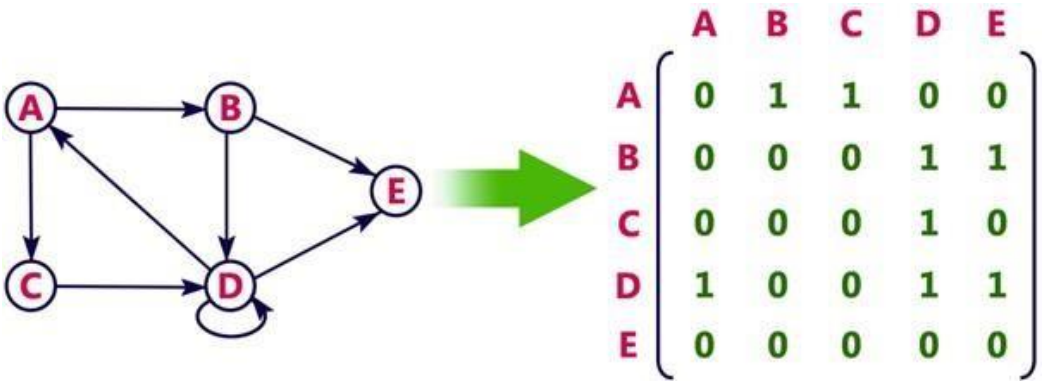
Consider the following **undirected graph representation**:

Undirected graph representation



Directed graph representation

See the directed graph representation:



Applications of Graph:

In **Computer science** graphs are used to represent the flow of computation.

- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example

of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.

- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.