PNS SCHOOL OF ENGINEERING & TECHNOLOGY

Nishamani Vihar, Marshaghai, Kendrapara

LECTURE NOTES ON DATA STRUCTURE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

3RD SEMESTER

PREPARED BY

MR. BISWARANJAN SWAIN

LECTURER IN COMPUTER SCIENCE & ENGINEERING

UNIT-1

INTRODUCTION

Data

The term ***Data*** is defined as a raw and unstructured fact that needs to be processed to make it meaningful. Data can be simple and unstructured at the same time until it is structured. Usually data contains facts, numbers, symbols, image, observations, perceptions, characters, etc.

Information

The term ***Information*** is defined as a set of data that is processed according to the given requirement in a meaningful way. To make the information useful and meaningful, it must be processed, presented and structured in a given context. Information is processed from data and possess context, purpose and relevance.

Data Type:

Data types are used within type systems, which offer different ways of defining, implementing and using the data. Different languages may use different terminology. Common data types are :

Integers,
Booleans,
Characters,
Floating-point numbers,
Alphanumeric strings.

Classes of data types

There are different classes of data types as given below.

1. Primitive data type
2. Composite data type
3. En-numerated data type
4. Abstract data type
5. Utility data type
6. Other data type

1. ***Primitive data types:***

All data in computers based on digital electronics is represented as bits (alternatives 0 and 1) on the lowest level. The smallest addressable unit of data is usually a group of bits called a byte (usually an octet, which is 8 bits). The unit processed by machine code instructions is called a word (as of 2011, typically 32 or 64 bits). Most instructions interpret the word as a binary number, such that a 32-bit word can represent unsigned integer values from 0 to $2^{32}-1$ or signed integer values from $-2^{31}$to $2^{31}-1$.

Boolean type :

The Boolean type represents the values true and false. Many programming languages do not have an explicit boolean type, instead interpreting (for instance) 0 as false and other values as true.

Numeric types such as:

The integer data types, or "whole numbers" may be subtype according to their ability to contain negative values (e.g. unsigned in C and C++).

Floating point data types, contain fractional values. They usually have predefined limits on both their maximum values and their precision. These are often represented as decimal numbers.

2. ***Composite / Derived data types :***

Composite types are derived from more than one primitive type. This can be done in a number of ways. The ways they are combined are called data structures. Composing a primitive type into a compound type generally results in a new type, e.g. array-of-integer is a different type to integer.

3. ***Enumerated Type :***

This has values which are different from each other, and which can be compared and assigned, but which do not necessarily have any particular concrete representation in the computer's memory; compilers and interpreters can
represent them arbitrarily.

String and text types such as:
Alphanumeric character. A letter of the alphabet, digit, blank space, punctuation mark, etc.
Alphanumeric strings, a sequence of characters. They are typically used to represent words and text.

Character and string types can store sequences of characters from a character set such as ASCII. Since most character sets include the digits, it is possible to have a numeric string, such as "1234".

4. ***Abstract data types***

Any type that does not specify an implementation is an abstract data type. For instance, a stack (which is an abstract type) can be implemented as an array (a contiguous block of memory containing multiple values), or as a linked list (a set of non-contiguous memory blocks linked by pointers).
Examples include:
- A queue is a first-in first-out list. Variations are Deque and Priority queue.
- A set can store certain values, without any particular order, and with no repeated values.
- A stack is a last-in, first out.
- A tree is a hierarchical structure.
- A graph.

Data structure

In Computer Science, data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently. Data structures determine the way in which information can be stored in computer and used. Finding the best data structure when solving a problem is an important part of programming. Programs that use the right data structure are easier to write, and work better.

Data Structure Operations:

The various operations that can be performed on different data structures are as follows:
1. Create– A data structure created from data.
2. Traverse – Processing each element in the list
3. Searching – Finding the location of given element.
4. Insertion – Adding a new element to the list.
5. Deletion – Removing an element from the list.
6. Sorting – Arranging the records either in ascending or descending order.
7. Merging – Combining two lists into a single list.
8. Modifying – the values of DS can be modified by replacing old values with new ones.
9. Copying – records of one file can be copied to another file.
10. Concatenating – Records of a file are appended at the end of another file.
11. Splitting – Records of big file can be splitting into smaller files.

Types of Data Structure :

Basically, data structures are divided into two categories:
- Linear data structure
- Non-linear data structure

Linear data structures:

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement. However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.
Popular linear data structures are:
1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.
2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.
It works just like a pile of plates where the last plate kept on the pile will be removed first.

3. Queue Data Structure

Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first. It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.

4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

Non linear data structures

Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Non-linear data structures are further divided into graph and tree based data structures.

1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.

2. Trees Data Structure

Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.

Abstract data type:

Abstract data types are like user-defined data types, which define the operations on the values using functions without specifying what is inside the function and how the operations are performed. It is a logical description of how we view the data and the operations that are allowed without knowing how they will be implemented. For example, an abstract stack could be defined by three operations:

1. push, that inserts some data item onto the structure,
2. pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and peek, that allows data on top of the structure to be examined without removal.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages.

Abstract Data Type in Computer Programming

In the course an abstract data type refers to a generalized data structure that accepts data objects stored as a list with specific behaviors defined by the methods associated with the underlying nature of the list.

Some common ADTs, are –

- Container
- Deque
- List
- Map
- Multimap
- Multiset
- Priority queue
- Queue
- Set
- Stack
- Tree
- Graph

Algorithm:

An algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning. An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

Complexity of Algorithm and Time, Space tradeoff:

The algorithms are evaluated by the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually, the
efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Best, worst and average case complexity:

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size n may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size n.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size n.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are
assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size n.

Time Complexity

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described asymptotically, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is $O(n^3)$.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Since an algorithm's performance time may vary with different inputs of the same size, one commonly uses the worst-case time complexity of an algorithm, denoted as T(n), which is defined as the maximum amount of time taken on any
input of size n. Time complexities are classified by the nature of the function T(n).

An algorithm with T(n) = O(n) is called a linear time algorithm, and an algorithm with $T(n) = O(2^n)$ is said to be an exponential time algorithm.

Space complexity

The way in which the amount of storage space required by an algorithm varies with the size of the problem it is solving. Space complexity is normally expressed as an order of magnitude, e.g. O(N^2) means that if the size of the problem (N) doubles then four times as much working storage will be needed.

Question Set:

1. What is ADT and explain ADT with example?
2. Define time complexity and space complexity.
3. Why it is necessary to find time and space complexity of an algorithm?
4. What are the possible operation on data structure?
5. Describe different type of data structure with example.

UNIT-2

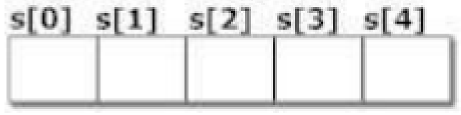
STRING PROCESSING

- A string in C is a array of character.
- It is a one dimensional array type of char.
- Every string is terminated by null character('\0') .
- The predefined functions gets() and puts() in C language to read and display string respectively.

Declaration of strings:

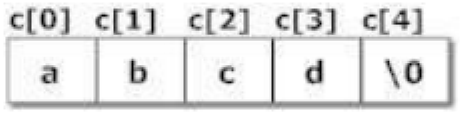
Strings are declared in C in similar manner as arrays. Only difference is that, strings are of char type.

Syntax:
char str_name[str_size];
Example:
char s[5];



Example:
char s[5];

Initialization of strings
char c[]="abcd";



String handling functions

- Many library function are defined under header file <string.h> to perform different tasks.
- Diffrent user defined functios are:
- Strlen()
- Strcpy()
- Strcmp()
- Strcat()

Strlen():

- The strlen( ) function is used to calculate the length of the string.
- It means that it counts the total number of characters present in the string which includes alphabets, numbers, and all special characters including blank spaces.

Strlen():

- The strlen( ) function is used to calculate the length of the string.
- It means that it counts the total number of characters present in the string which includes alphabets, numbers, and all special characters including blank spaces.

Example:
char str[] = "Learn C Online";
int strLength;
strLength = strlen(str); //strLength contains the length of the string i.e. 14

Strcpy()

• strcpy function copies a string from a source location to a destination location and provides a null character to terminate the string.

Syntax:

```
strcpy(Destination_String,Source_String);
```

Example:

```
char *Destination_String;
char *Source_String = "Learn C Online";
strcpy(Destination_String,Source_String);
printf("%s", Destination_String);
```

Output:

```
Learn C Online
```

Strcmp()

• Strcmp() in C programming language is used to compare two strings.

• If both the strings are equal then it gives the result as zero but if not then it gives the numeric difference between the first non matching characters in the strings.

Syntax:

```
int strcmp(string1, string2);
```

Example:

```
char *string1 = "Learn C Online";
char *string2 = "Learn C Online";
int ret;
ret=strcmp(string1, string2);
printf("%d",ret);
```

Output:

```
0
```

Strcat()

• The strcat() function is used for string concatenation in C programming language. It means it joins the two strings together Syntax:

```
strncat(Destination_String, Source_String,no_of_characters);
```

Example:

```
char *Destination_String="Visit ";
char *Source_String = "Learn C Online is a great site";
strncat(Destination_String, Source_String,14);
puts( Destination_String);
```

Output:

```
Visit Learn C Online
```

Program to check wheaher a word is polyndrome or not

```
#include<stdio.h>
#include<string.h>
void main()
{
char s1[10], s2[10];
int x;
gets(s1);
strcpy(s2,s1);
strrev(s1);
x=strcmp(s1,s2);
if(x==0)
printf("pallindrome");
else
printf("not pallindrome");
getch();}
```

UNIT-3

ARRAYS

Definition :

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.

For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject, instead of that, we can define an array which can store the marks in each subject at contiguous memory locations.

The array marks[10] defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. marks[0] denotes the marks in first subject, marks[1] denotes the marks in 2nd subject and so on
Array is a list of finite number of n homogeneous data elements i.e. the elements of same data types Such that:

- The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.

- The elements of the array are stored respectively in the successive memory locations.

The number n of elements is called length or size of array. If not explicitly stated, we will assume the index set consists of integers 1, 2, 3 …n. In general the length or the number of data elements of the array can be obtained from the index set by the formula

Length= UB – LB + 1
Where UB is the largest index, called the upper bound, and LB is the smallest index,

called the lower bound. Note that length = UB when LB = 1.

The elements of an array A are denoted by subscript notation à a1, a2, a3…an Or by the

parenthesis notation -> A (1), A (2),…., A(n)

Or by the bracket notation -> A[1], A[2],….,A[n].

We will usually use the subscript notation or the bracket notation.

REPRESENTATION OF LINEAR ARRAYS IN MEMORY:

Let LA is a linear array in the memory of the computer. The memory of computer is simply a sequence of addressed locations.
LOC (LA[k]) = address of element LA[k] of the array LA.
As previously noted, the elements of LA are stored in the successive memory cells.
Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by Base (LA)
And called the base address of LA.

Using base address the computer calculates the
address of any element of LA by the following formula:

LOC (LA[k]) = Base (LA) + w (k-lower bound)

Where w is the number of words per memory cell for the array LA.

OPERATIONS ON ARRAYS :

Various operations that can be performed on an array

- Traversing
- Insertion
- Deletion
- Sorting
- Searching
- Merging

TRAVERSING LINEAR ARRAY:

Let A be a collection of data elements stored in the memory of the computer.
Suppose we want to print the content of each element of A or suppose we want to count the number of elements of A, this can be accomplished by traversing A, that is, by accessing and processing each element of a exactly ones.
The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures, such as linked list, can also be easily traversed. On the other hand, traversal of nonlinear structures, such as trees and graph, is considerably more complicated.

Algorithm: (Traversing a Linear Array)
Here LA is a linear array with lower bound LB and upper bound UB.
This algorithm traverses LA applying an operation PROCESS to each element of LA.

```
1. [Initialize counter] Set k: =LB.
2. Repeat steps 3 and 4 while k <=UB.

3. [Visit Element] Apply PROCESS to LA [k].

4. [Increase Counter] Set k: =k + 1.  [End of step 2 loop]

5. Exit.
```

INSERTION AND DELETION IN LINEAR ARRAY:

Let A be a collection of data elements in the memory of the computer.
Inserting refers to the operation of adding another element to the collection A
Deleting refers to the operation of removing one element from A.
Inserting an element at the end of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new location to accommodate the new elements and keep the order of the other elements.
Similarly, deleting an element at the end of the array presents no difficulties, but deleting the element somewhere in the middle of the array requires that each subsequent element be moved one location upward in order to fill up the array.
The following algorithm inserts a data element ITEM in to the Kth position in the linear array LA with N elements.

Algorithm for Insertion: (Inserting into Linear Array)

INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K<=N.
The algorithm inserts an element ITEM into the Kth position in LA.

```
1. [Initialize counter] Set J: = N.
2. Repeat Steps 3 and 4 while j >= k;
3. [Move jth element downward.] Set LA [J + 1]: =LA [J].
4. [Decrease counter]
Set J: = J-1  [End of step 2 loop]
5. [Insert element] Set LA [K]:=ITEM.
6. [Reset N] Set N:=N+1
7. EXIT.
```

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

Algorithm for Deletion: (Deletion from a Linear Array)

DELETE (LA, N, K, ITEM)

Here LA is a Linear Array with N elements and K is the positive integer
such that K<=N. This algorithm deletes the Kth element from LA.

```
1. Set ITEM: = LA [k].
2. Repeat for J = K to N – 1.
[Move J + 1$^{st}$ element upward] Set LA [J]: = LA [J +1].
 [End of loop]
3. [Reset the number N of elements in LA] Set N: = N-1
4. EXIT
```

MULTIDIMENSIONAL ARRAY:

1. Array having more than one subscript variable is called Multi- Dimensional array.
2. Multi Dimensional Array is also called as Matrix.

Consider the Two dimensional array -

1. Two Dimensional Array requires Two Subscript Variables
2. Two Dimensional Array stores the values in the form of matrix.
3. One Subscript Variable denotes the —Row of a matrix.
4. Another Subscript Variable denotes the —Column of a matrix.

Declaration and Use of Two Dimensional Array :

```
int a[3][4];
```

Use :

```
for(i=0;i<row;i++)
{
      for(j=0;j<col;j++)
     {
           printf("%d",a[i][j]);
     }
}
```

Meaning of Two Dimensional Array :

1. Matrix is having 3 rows (i takes value from 0 to 2)
2. Matrix is having 4 Columns (j takes value from 0 to 3)
3. Above Matrix 3×4 matrix will have 12 blocks having 3 rows & 4 columns.
4. Name of 2-D array is _a_ and each block is identified by the row & column Number.
5. Row number and Column Number Starts from 0.

Cell Location	Meaning
a[0][0]	0th Row and 0th Column
a[0][1]	0th Row and 1st Column
a[0][2]	0th Row and 2nd Column
a[0][3]	0th Row and 3rd Column
a[1][0]	1st Row and 0th Column
a[1][1]	1st Row and 1st Column
a[1][2]	1st Row and 2nd Column
a[1][3]	1st Row and 3rd Column
a[2][0]	2nd Row and 0th Column
a[2][1]	2nd Row and 1st Column
a[2][2]	2nd Row and 2nd Column
a[2][3]	2nd Row and 3rd Column

Two-Dimensional Array : Summary with Sample Example

Summary Point	Explanation
No of Subscript Variables Required	2
Declaration	a[3][4]
No of Rows	3
No of Columns	4
No of Cells	12
No of for loops required to iterate	2

MEMORY REPRESENTATION :

1. 2-D arrays are Stored in contiguous memory location row wise.
2. X 3 Array is shown below in the first Diagram.
3. Consider 3×3 Array is stored in Contiguous memory location which starts from 4000 .
4. Array element a[0][0] will be stored at address 4000 again a[0][1] will be stored to next memory location i.e Elements stored row-wise
5. After Elements of First Row are stored in appropriate memory location , elements of next row get their corresponding memory locations.
6. This is integer array so each element requires 2 bytes of memory.

Array Representation:

- Column-major
- Row-major

Arrays may be represented in Row-major form or Column-major form.

In **Row- major** form, all the elements of the first row are printed, then the elements of the second row and so on up to the last row.

In **Column-major** form, all the elements of the first column are printed, then the elements of the second column and so on up to the last column.

The ‗C‘ program to input an array of order m x n and print the array contents in row major and column major is given below. The following array elements may be entered during run time to test this program:
Output:
Row Major:

```
1  2  3

4  5  6

7  8  9
```

Column Major:

```
1  4  7

2  5  8

3  6  9
```

Basic Memory Address Calculation :

a[0][1] = a[0][0] + Size of Data Type

Element	Memory Location
a[0][0]	4000
a[0][1]	4002
a[0][2]	4004
a[1][0]	4006
a[1][1]	4008
a[1][2]	4010
a[2][0]	4012
a[2][1]	4014
a[2][2]	4016

Array and Row Major, Column Major order arrangement of 2 d array :

An array is a list of a finite number of homogeneous data elements. The number of elements in an array is called the array length. Array length can be obtained from the index set by the formula

Length=UB-LB+1

Where UB is the largest index , called the upper bound and LB is the smallest index , called the lower bound. Suppose int Arr[10] is an integer array. Upper bound of this array is 9 and lower bound of this array is 0 ,so the length is 9- 0+1=10.

In an array ,the elements are stored successive memory cells. Computer does not need to keep track of the address of every elements in memory. It will keep the address of the first location only and that is known as base address of an array.

Using the base address , address of any other location of an array can be calculated by the computer. Suppose Arr is an array whose base address is Base(Arr) and w is the number of memory cells required by each elements of the array. The address of Arr[k] – k being the index value can be obtained by Using the formula

Address(Arr[k])=Base(Arr)+w(k-LowerBound)

2 d Array :- Suppose Arr is a 2 d array. The first dimension of Arr contains the index set 0,1,2, … row-1 (the lower bound is 0 and the upper bound is row-1) and the second dimension contains the index set 0,1,2,… col-1(with lower bound 0 and upper bound col-1.)

The length of each dimension is to be calculated .The multiplied result of both the lengths will give you the number of elements in the array.

Let‘s assume Arr is an two dimensional 2 X 2 array .The array may be stored in memory one of the following way :-

- Column by columni,e column major order
- Row by row , i,e in row major order. The following figure shows both representation of the above array.

By row-major order, we mean that the elements in the array are so arranged that the subscript at the extreme right varies fast than the subscript at it‘s left., while in column-major order , the subscript at the extreme left changes rapidly ,Then the subscript at it‘s right and so on.

1,1

2,1

1,2

2,2

Column Major Order

1,1

1,2

2,1

2,2

Row major order

Now we know that computer keeps track of only the base address. So the address of any specified location of an array , for example Arr[j,k] of a 2 d array Arr[m,n] can be calculated by using the following formula :- (Column major order)

Address(Arr[j,k])= base(Arr)+w[m(k-1)+(j-1)]

(Row major order)

Address(Arr[j,k])=base(Arr)+w[n(j-1)+(k-1)]

For example

Arr(25,4) is an array with base value 200.w=4 for this array. The address of Arr(12,3) can be calculated using row-major order as

Address(Arr(12,3))=200+4[4(12-1)+(3-1)]
=200+4[4*11+2]
=200+4[44+2]
=200+4[46]
=200+184
=384
Again using column-major order
Address(Arr(12,3))=200+4[25(3-1)+(12-1)]
=200+4[25*2+11]
=200+4[50+11]
=200+4[61]
=200+244
=444

SPARSE MATRIX :

Matrix with relatively a high proportion of zero entries are called sparse matrix. Two general types of n-square sparse matrices are there which occur in various applications are mention in figure below(It is sometimes customary to omit blocks of zeros in a matrix as shown in figure below)

$$\begin{array}{cccc}
4 & & & \\
3 & -5 & & \\
1 & 0 & 6 & \\
-7 & 8 & -1 & 3 \\
5 & -2 & 0 & -8
\end{array}
\qquad\qquad
\begin{array}{ccccc}
5 & -3 & & & \\
1 & 4 & 3 & & \\
 & 9 & -3 & 6 & \\
 & & 2 & 4 & -7 \\
 & & & 3 & 0
\end{array}$$

Triangular matrix Tridiagonal matrix

Triangular matrix

This is the matrix where all the entries above the main diagonal are zero or equivalently where non-zero entries can only occur on or below the main diagonal is called a (lower)Triangular matrix.

Tridiagonal matrix

This is the matrix where non-zero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a Tridiagonal matrix. The natural method of representing matrices in memory as two-dimensional arrays may not be suitable for sparse matrices i.e. one may save space by storing only those entries which may be non-zero.

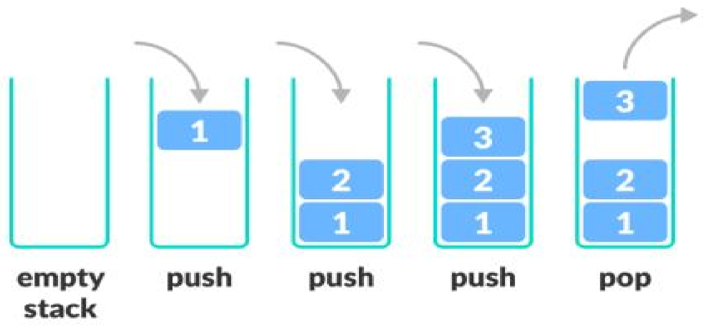
UNIT-4

STACKS & QUEUES

STACK

 Stack is a linear data structure in which an element may be inserted or deleted at one end called TOP of the stack.

 That means the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.



Stack Push and Pop Operations

 Stack is called LIFO (Last-in-first-Out) Str. i.e. the item just inserted isdeleted first.

 There are 2 base operations associated with stack

1. Push :- This operation is used to insert an element into stack.
2. Pop :- This operation is used to delete an element from stack.

Condition also arise :

1. Overflow :- When a stack is full and we are attempting a push operation , overflow condition arises.
2. Underflow :- When a stack is empty , and we are attempting a pop operation then underflow condition arises.

Representation of Stack in Memory :

A stack may be represented in the memory in two ways:

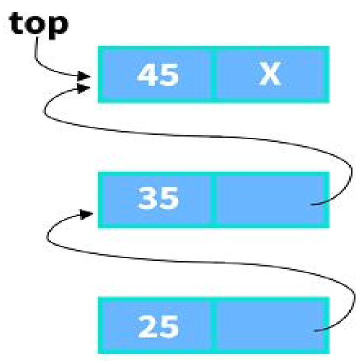
1. Using one dimensional array i.e. Array representation of Stack.
2. Using single linked list i.e. Linked list representation of stack.

Array Representation of Stack :

To implement a stack in memory, we need a pointer variable called TOP that hold the index of the top element of the stack, a linear array to hold the elements of the stack and a variable MAXSTK which contain the size of the stack.

Linked List Representation of Stack :

 Array representation of Stack is very easy and convenient but it allows only to represent fixed sized stack.

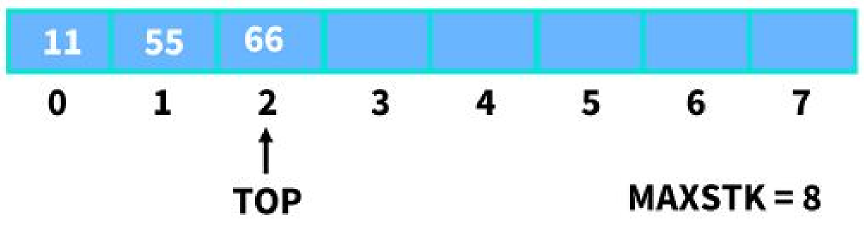


- But in several application size of the stack may very during program execution, at that cases we represent a stack using linked list.
- Single linked list is sufficient to represent any Stack.
- Here the 'info' field for the item and 'next' field is used to print the next item.

INSERTION IN STACK (PUSH OPERATION)

This operation is used to insert an element in stack at the TOP of the stack.

```
Algorithm :-
PUSH (STACK, TOP, MAXSTK, ITEM)
This procedure pushes an item into stack.
1. If TOP = MAXSTK then print Overflow and Return.
2. Set TOP = TOP+1 (Increase TOP by 1)
```

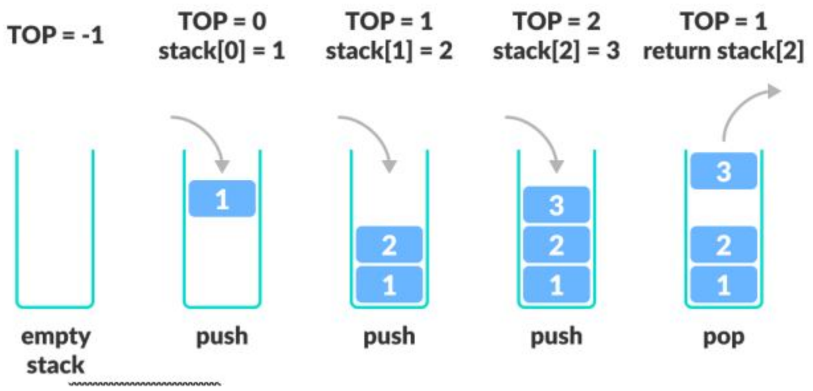


```
3. Set STACK[TOP] =  ITEM (Inserts ITEM in new TOP position)
4. Return
```

(Where Stack = Stack is a list of linear structure.
TOP = printer variable which contain the location of TOP element of the stack.
MAXSTK = A variable which contain size of stack
ITEM = Item to be inserted)

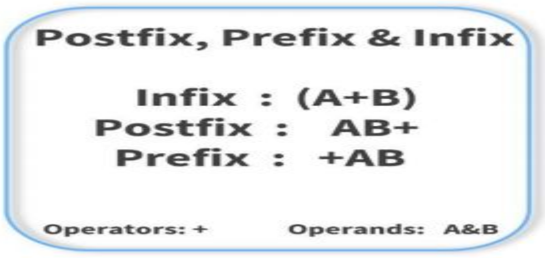
DELETION IN STACK (POP OPERATION)

This operation is used to delete an element from stack.



Algorithm :-
POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

```
1. If TOP = 0 the print Underflow and Return.
2. Set ITEM = STACK[TOP] (Assigns TOP element to ITEM)
3. Set TOP = TOP-1 (Decreases TOP by 1)
Working of Stack Data Structure
5.      Return
```



ARITHMETIC EXPRESSION
There are 3 notation to represent an arithmetic expression.

1. Infix notation
2. Prefix notation
3. Postfix notation

INFIX NOTATION
The conventional way of writing an expression is called as infix.
Ex: A+B, C+D, E*F, G/M etc.
Here the notation is
<Operand><Operator><Operand>
This is called infix because the operators come in between the operands.

PREFIX NOTATION
This notation is also known as "POLISH NOTATION"
Here the notation is
<Operator><Operand><Operand>
Ex: +AB, -CD, *EF, /GH
POSTFIX NOTATION
This notation is called as postfix or suffix notation where operator is suffixed by operand.
Here the notation is
<Operand ><Operand><Operator>
Ex: AB+, CD-, EF*, GH/
This notation is also known as " REVERSE POLISH NOTATION."
CONVERSION FROM INFIX TO POSTFIX EXPRESSION

Algorithm:
POLISH(Q,P)
Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

```
1. Push " ( " onto stack and add " ) " to the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q
until the STACK is empty.
3. If an operand is encountered, add it top.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator X is encountered, then:
```

a) Repeatedly POP from STACK and add to P each operator (on the top of the STACK) which has the same precedence as or higher precedence than the operator X .
b) Add the operator X to STACK.
6. If a right parenthesis is encountered then:
a) Repeatedly POP from STACK and add to P each operator (on the top of the STACK) until a left parenthesis is encountered.
b) Remove the left parenthesis.
[End of if str.]
[End of Step-2 Loop]
7. Exit

Ex: A+(B+C – (D/E+F)*G)*H)

Symbol Scanned	STACK	EXPRESSION (POSTFIX)
	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
C	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)		ABC*DEF^/G*-H*+

Equivalent Postfix Expression ->ABC*DEF^/G*-H*+

Algorithm:
EVALUATION OF POSTFIX EXPRESSION
This algorithm finds the value of an arithmetic expression P written in Postfix notation.
1. Add a right parenthesis ") " at the end of P.
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the ") " is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator X is encountered then:
a) Remove the two top element of STACK, where A is top element and B is the next-to-top element.
b) Evaluate B X A.

c) Place the result of (b) on STACK.
[End of if str.]
5. Set value equal to the top element on STACK.
6. Exit

Ex: 5, 6, 2, +, *, 12, 4, /, -)

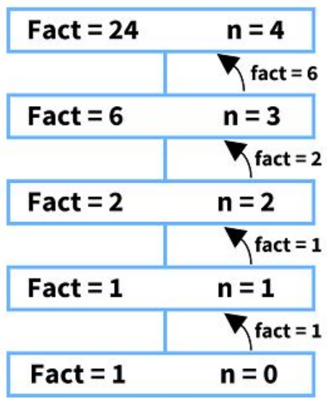
Symbol Scanned	STACK
5	5
6	5,6
2	5,6,2
+	5,8
*	40
12	40,12
4	40,12,4
/	40,3
-	37 (Result)
)	

APPLICATIONS OF STACK:

- Recursion process uses stack for its implementation.
- Quick sort uses stack for sorting the elements.
- Evaluation of antithetic expression can be sone by using STACK.
- Conversion from infix to postfix expression
- Evaluation of postfix expression
- Conversion from infix to prefix expression
- Evaluation of prefix expression.
- Backtracking
- Keeptrack of post-visited (history of a web- browsing)

IMPLEMENTATION OF RECURSION
A function is said to be Recursive function if it call itself or it call a function such that the function call back the original function.

This concept is called as Recursion.
The Recursive function has two properties.
I. The function should have a base condition.
II. The function should come closer towards the base condition.
The following is one of the example of recursive function which is described below.



Factorial of a no. using Recursion

The factorial of a no. ‘n’ is the product of positive integers from 1 to n.

n ! = 1 X 2 X 3 X . . . . .X (n-) X n

Physically proved n ! = n X (n-1) !

The factorial function can be defined as follows.

I. If n=0 then n ! = 1

II. If n>0 then n ! = n.(n-1) !

The factorial algorithm is given below factorial (fact , n)

This procedure calculates n! and returns the value in the variable fact.

1. If n=0 then fact=1 and Return.
2. Call factorial (fact, n-1)
3. Set fact = fact*n
4. Return

Ex: Calculate the factorial of 4.

4 ! = 4 x 3 !

3 ! = 3 x 2 !

1 ! = 1 x 1 = 1

2 ! = 2 x 1 = 2

3 ! = 3 x 2 = 6

4 ! = 4 x 6 = 24

2 ! = 2 x 1 !

1 ! = 1 x 0 !

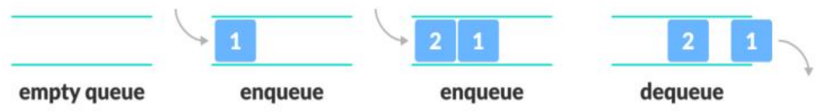
0 ! = 1

Queue :

 Queue is a linear data structure or sequential data structure where insertion take place at one end and deletion take place at the other end.

 The insertion end of Queue is called rear end and deletion end is called front end.

 Queue is based on (FIFO) First in First Out Concept that means the node i.e. added first is processed first.



 HereEnqueue is Insert Operation.

FIFO Representation of Queue

Dequeue is nothing but Delete Operation.

Types of Queue:

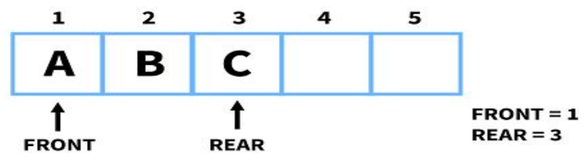
1. General Queue
2. Circular Queue
3. Deque
4. Priority Queue
5.

REPRESENTATION OF QUEUE IN MEMORY

- The Queue is represented by a Linear array "Q" and two pointer variable FRONT and REAR.
- FRONT gives the location of element to be deleted and REAR gives the location after which the element will be inserted.
- The deletion will be done by setting Front = Front + 1
- The insertion will be done by setting Rear = Rear + 1

INSERTION IN QUEUE(Enqueue):

Ex:

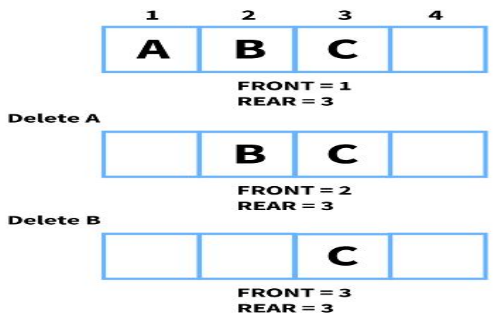


Algorithm:

Insert (Q, ITEM, Front, Rear)

This procedure insert ITEM in queue Q.

```
1. If Front = 1 and Rear = Max
2. Print 'Overflow' and Exit.
3. If front = NULL than
4. Front = 1
5. Rear = 1
6. else
7. Rear = Rear +1
8. 3. Q [Rear] = ITEM
9. 4. Exit
```



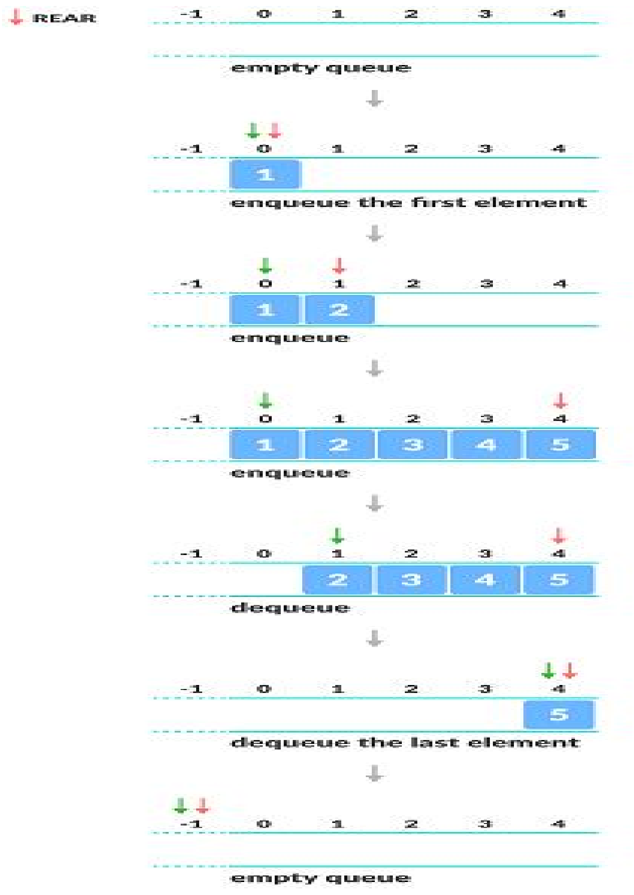
DELETION IN QUEUE(Dequeue):

Ex:

Algorithm:

```
Delete (Q, ITEM, FRONT, REAR)
This procedure remove element from queue Q.
1.  If Front = Rear = NULL
Print ‘Underflow’ and Exit.
2.  ITEM = Q (FRONT)
3.  If Front = Rear
Front = NULL
Rear = NULL
else
Front = Front + 1
4.  Exit
```

Enqueue and Dequeue Operations:



PRIORITY QUEUE

 If is a type of queue in which each element has been assigned a priority such that the order in which the elements are processed according to the elements are processed according to the following rule

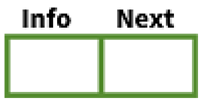
I. This element of high priority is processed first.

II. The element having same priority are processed according to the order in which they are inserted.

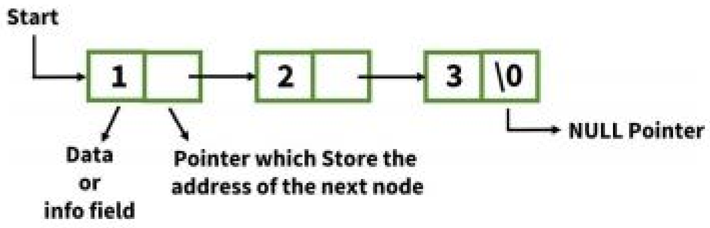
 The lower ranked no. enjoy high priority.

LINKED LIST

- Linked List is a collection of data elements called as nodes.
- The node has 2 parts
1. Info is the data part
2. Next i.e. the address part that means it points to the next node.



- If linked list adjacency between the elements are maintained by means of links or pointers.
- A link or pointer actually the address of the next node.



- The last node of the list contain '\0' NULL which shows the end of the list.
- The linked list contain another pointer variable 'Start' which contain the address the first node.
- Linked List is of 4 types
1. Single Linked List
2. Double Linked List
3. Circular Linked List
4. Header Linked List

Representation of Linked List in Memory
- Linked List can be represented in memory by means 2 linear arrays i.e. Data or info and Link or address.
- They are designed such that info[K] contains the Kth element and Link[K] contains the next pointer field i.e. the address of the Kth element in the list.
- The end of the List is indicated by NULL pointer i.e. the pointer field of the last element will be NULL or Zero.

SINGLE LINKED LIST

A Single Linked List is also called as one-way list. It is a linear collection of data elements called nodes, where the linear order is given by means of pointers. Each node is divided into 2 parts.

I. Information
II. Pointer to next node.

OPERATION ON SINGLE LINKED LIST

1. TRAVERSAL

```
Algorithm:
Display (Start) This algorithm traverse the list starting from the 1st
 node to the
end of the list.
Step-1 : "Start" holds the address of the first node.
Step-2 : Set Ptr = Start [Initializes pointer Ptr]
Step-3 : Repeat Step 4 to 5, While Ptr ≠ NULL
```

```
Step-4  : Process  info[Ptr]  [apply Process  to info(Ptr)]
Step-5 : Set Ptr = next[Ptr] [move Print to next node]
[End of loop]
Step-6 : Exit
```

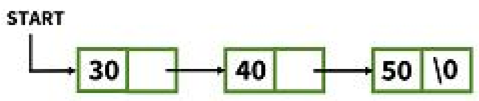
2. INSERTION

The insertion operation is used to add an element in an existing Linked List.
There are various position where node can be inserted.

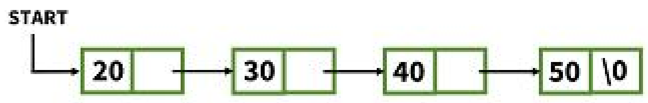
a) Insert at the beginning
b) Insert at end
c) Insert at specific location.

INSERT AT THE BEGINNING

Suppose the new mode whose information field contain 20 is inserted as the first node.



(before insertion)



(After insertion)

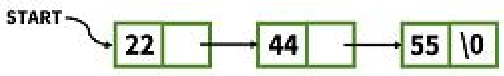
Algorithm:
This algorithm is used to insert a node at the beginning of the Linked List.
Start holds the address of the first node.

```
Step-1 : Create a new node named as P.
Step-2 : If P == NULL then print “Out of memory space” and exit.
Step-3 : Set info [P] =   (copies a new data into a new node)
Step-4 : Set next [P] = Start (new node now points to original first node)
Step-5 : Set Start = P
Step-6 : Exit
```

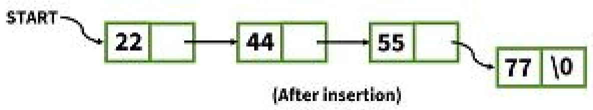
INSERT AT THE END

 To insert a node at the end of the list, we need to traverse the List and advance the pointer until the last node is reached.

 Suppose the new node whose information field contain 77 is inserted at the last node.



(before insertion)

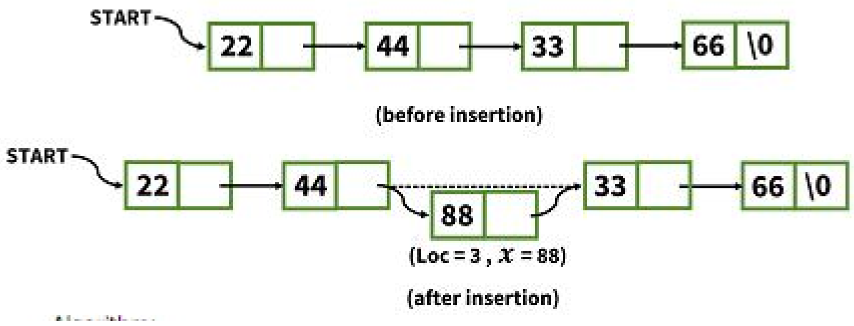


(After insertion)

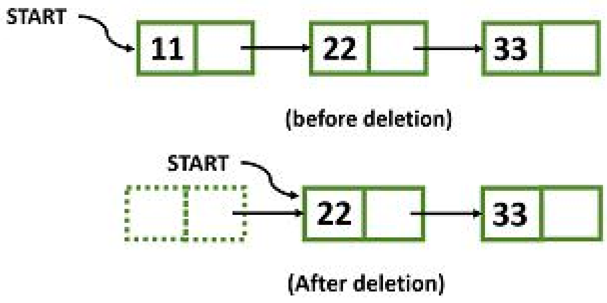
Algorithm:
This algorithm is used to insert a node at the end of the linked list.
‘Start’ holds the address of the first node.

```
Step-1 : Create a new node named as P.
Step-2 : If P = NULL then print “Out of memory space” and Exit.
Step-3 : Set info [P] =  (copies a new data into a new node)
Step-4 : Set next [P] = NULL
Step-5 : Set Ptr = Start
Step-6 : Repeat Step-7 while Ptr ≠ NULL
```

Step-7 : Set temp = Ptr
Ptr = next [Ptr]
(End of step-6 loop)
Step-8 : Set next [temp] = P
Step-9 : Exit
INSERT AT ANY SPECIFIC LOCATION
To insert a new node at any specific location we scan the List from the beginning and move up to the desired node where we want to insert a new node. In the below fig. Whose information field contain 88 is inserted at 3rd location.



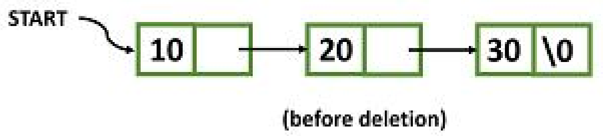
Algorithm:
'Start' holds the address of the 1st
 node.
Step-1 : Set Ptr = Start
Step-2 : Create a new node named as P.
Step-3 : If P = NULL then write 'Out of memory' and Exit.
Step-4 : Set info [P] = (copies a new data into a new node)
Step-5 : Set next [P] = NULL
Step-6 : Read Loc
Step-7 : Set i = 1
Step-8 : Repeat steps 9 to 11 while Ptr ≠ NULL and i < Loc
Step-9 : Set temp = Ptr.
Step-10 : Set Ptr = next [Ptr]
Step-11 : Set i = i + 1
[End of step-7 loop]
Step-12 : Set next [temp] = P
Step-13 : Set next [P] = Ptr
Step-14 : Exit
3. DELETION
The deletion operation s used to delete an element from a single linked list. There are various position where node can be deleted.
a) Delete the 1st
 Node

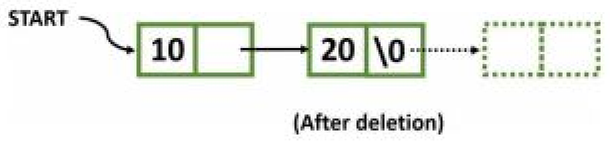


Algorithm:
Start holds the address of the 1st
 node.

```
Step-1 : Set temp = Start
Step-2 : If Start = NULL then write 'UNDERFLOW' & Exit.
Step-3 : Set Start = next[Start]
Step-4  : Free the space associated with temp.
Step-5 : Exit
```

b) Delete the last node

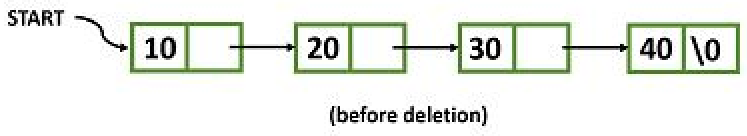


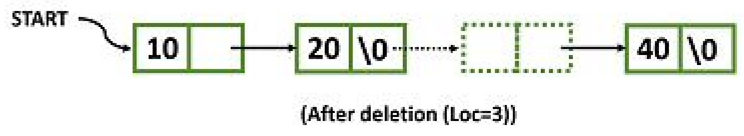


Algorithm:
Start holds the address of the 1st
 node.

```
Step-1 : Set Ptr = Start
Step-2 : Set temp = Start
Step-3 : If Ptr = NULL then write 'UNDERFLOW' & Exit.
Step-4 : Repeat Step-5 and 6 While next[Ptr] ≠ NULL
Step-5 : Set temp = Ptr
Step-6 : Set Ptr = next[Ptr]
(End of step 4  loop)
Step-7 : Set next[temp] = NULL
Step-8 : Free the space associated with Ptr.
Step-9 : Exit
```

c) Delete the node at any specific location

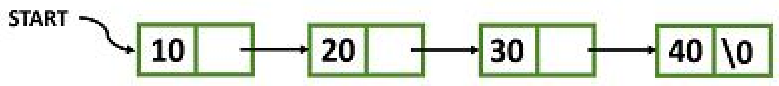




Algorithm:
Start holds the address of the 1st
 node.

```
Step-1 : Set Ptr = Start
Step-2 : Set temp = Start
Step-3 : If Ptr = NULL then write 'UNDERFLOW' and Exit.
Step-4 : Set i = 1
Step-5 : Read Loc
Step-6 : Repeat Step-7 to 9 while Ptr ≠ NULL and i < Loc
Step-7 : Set temp = Ptr
Step-8 : Set Ptr = next[Ptr]
Step-9 : Set i = i+1
(End of Step 6 loop)
Step-10  : Set next[temp] =  next[Ptr]
Step-11 : Free the space associated with Ptr.
Step-12 : Exit
```

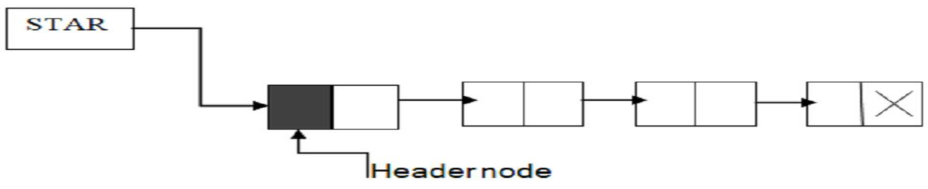
4. SEARCHING
Searching means finding an element from a given list.



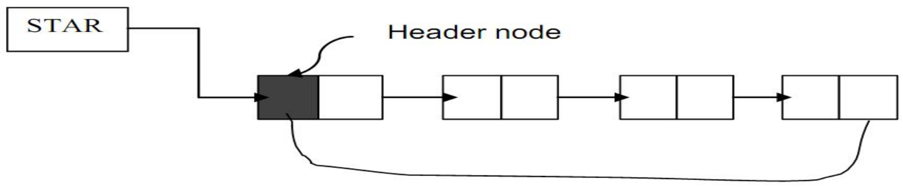
Algorithm:
Start holds the address of the 1st
node.

```
Step-1 : Set Ptr = Start
Step-2 : Set Loc = 1
Step-3 : Read element
Step-4 : Repeat Step-5 and 7 While Ptr ≠ NULL
Step-5 : If element = info[Ptr] then Write ‘Element found at position’, Loc and
Exit.
Step-6 : Set Loc = Loc+1
Step-7 : Set Ptr = next[Ptr]
(End of step 4 loop)
Step-8 : Write ‘Element not found’
Step-9 : Exit
```

Header linked list:-
A header linked list is a special type of linked list. Which always contains a special node called header node at the beginning of the list so in a header linked list will not point to first node of the list. But start will contain the address of the header node.
There are two types of header linked list i.e. Grounded header linked list & Circular header linked list.
Grounded header linked list:-
It is a header linked list where last node contains the NULL pointer. LINK [start] = NULL indicates that a grounded header linked list is empty.



Circular header linked list:-
It is a header linked list where the last node points back to the header node.



LINK [START] = START is indicates that a circular linked list is empty.
Circular header list are frequently used instead of ordinary linked list because many operation are much easier to implement header list.
This comes from the following two properties, all circular header lists.
➢ The NULL pointer is not used & hence contains valid address.
Every ordinary node has a predecessor. So the first node may not required a special case

UNIT-6

TREE

Tree - Terminology

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows...

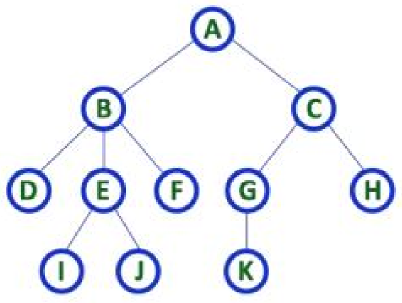
Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively. In tree data structure, every individual element is called as Node. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

Example



Tree with 11 nodes and 10 edges.

In any tree with n nodes there will be n-1 edges.

In a tree every individual element is called as NODE.

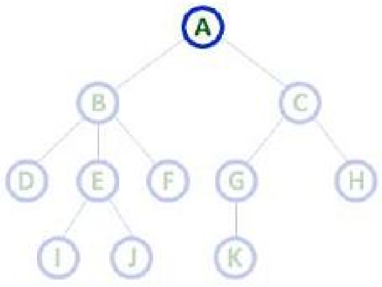
Terminology:

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

1. *Root*

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

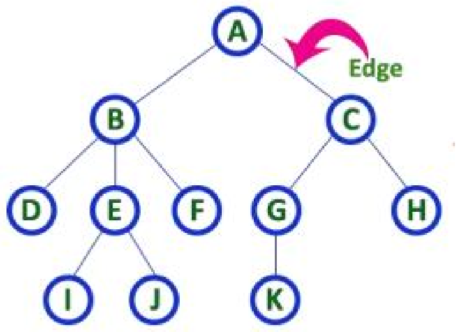
Here A is the root node. In any tree the first node is called as **root** node



2. Edge

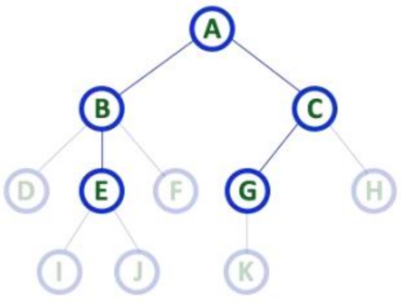
In a tree data structure, the connecting link between any two nodes is called as EDGE.

In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



3. Parent

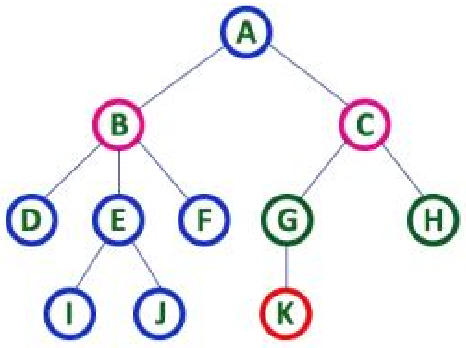
In a tree data structure, the node which is a predecessor of any node is called as PARENT NODE. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "The node which has child / children".



In the above figure A,B, C, G, E are parent nodes.

4. Child

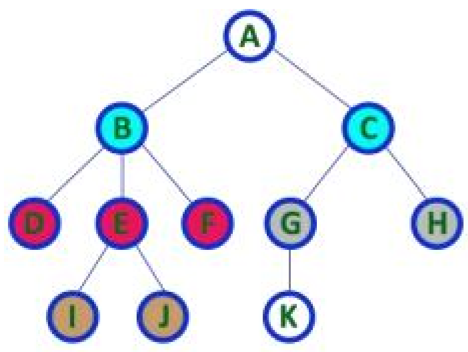
In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are children of A
G & H are children of C
K is child of G

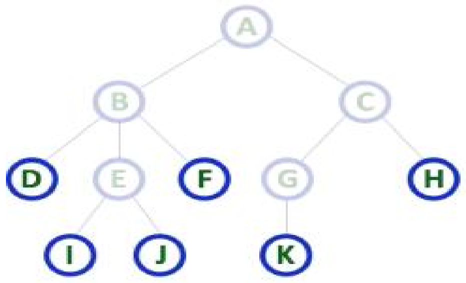
5. Siblings

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



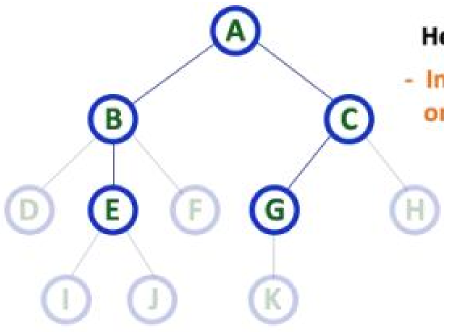
Here B & C are siblings.
D, E & F are siblings.

6. Leaf
In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.



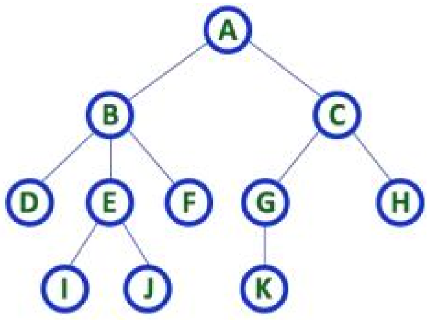
Here D, F, H, I, J, K are leaf nodes.

7. Internal Nodes
In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.
In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



Here A, B, C, E, G are internal nodes.

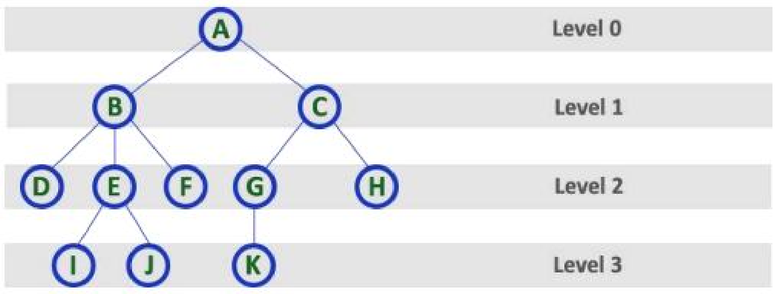
8. Degree
In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



Here degree of B is 3
Here degree of A is 2
Here degree of F is 0

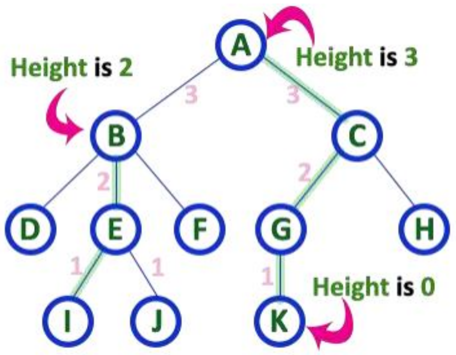
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

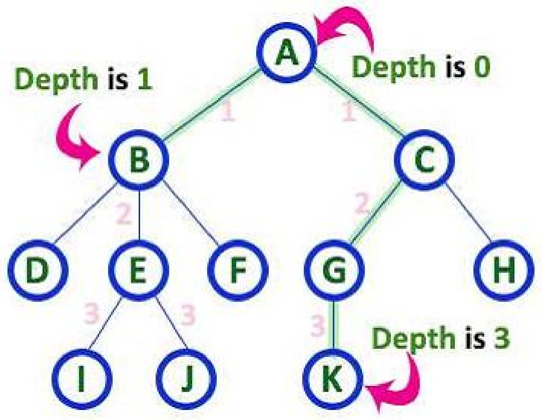
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



Here height of tree is 3.

11. Depth

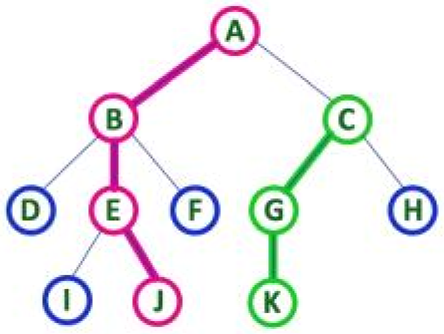
In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree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



Here depth of tree is 3.

12. Path

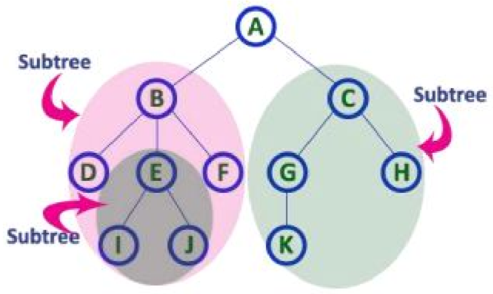
In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



Here path between A & J is: A-B-E-J
Here C & K is : C-G-K

13. Sub Tree

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



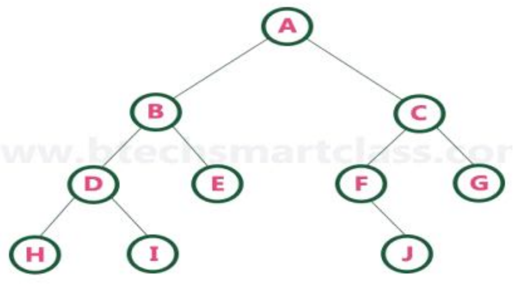
Binary Tree Datastructure:

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



PROPERTIES OF BINARY TREE: Some of the important properties of a binary tree are as follows:
1. If h = height of a binary tree, then

a. Maximum number of leaves = 2h

b. Maximum number of nodes = 2h + 1 – 1

2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.

3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2l node at level l.

4. The total number of edges in a full binary tree with n node is n - 1

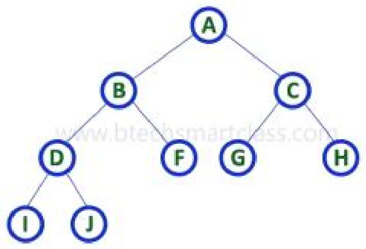
There are different types of binary trees and they are...

1. Strictly Binary Tree

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

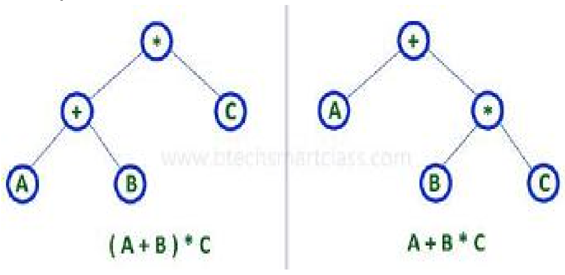
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

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



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

Example



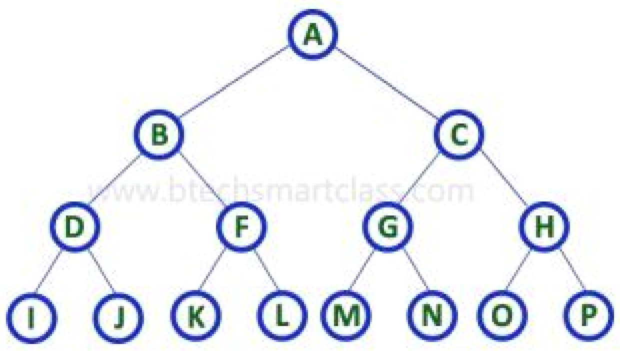
2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2level number of nodes. For example at level 2 there must be 22 = 4 nodes

and at level 3 there must be 23 = 8 nodes.

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.

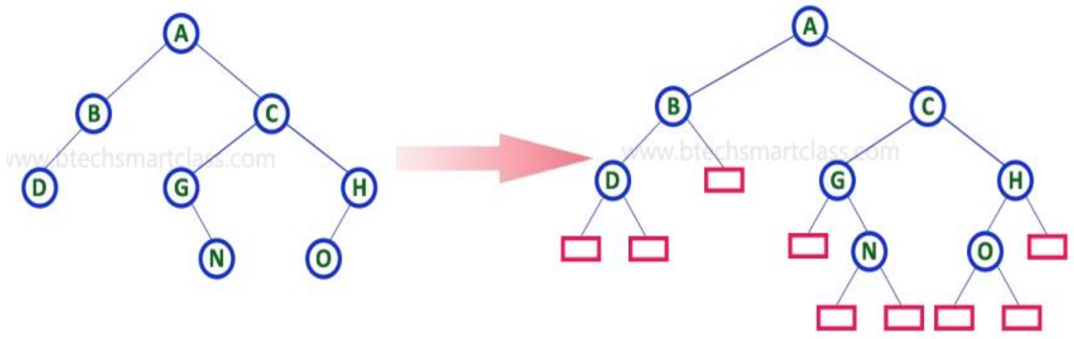
Complete binary tree is also called as Perfect Binary Tree



3. Extended Binary Tree

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

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



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

Traversal is a process to visit all the nodes of a tree and may print their values too.

Because, all nodes are connected via edges links we always start from the root head node. That is, we cannot random access a node in tree. There are three ways which we use to traverse a tree –

In-order Traversal

Pre-order Traversal

Post-order Traversal

Generally we traverse a tree to search or locate given item or key in the tree or to print all the values it contains.

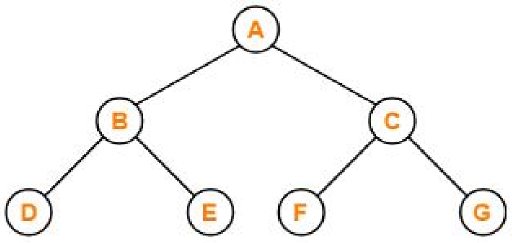
1. *Preorder Traversal-*

Algorithm-

1. Visit the root
2. Traverse the left sub tree i.e. call Preorder (left sub tree)
3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right

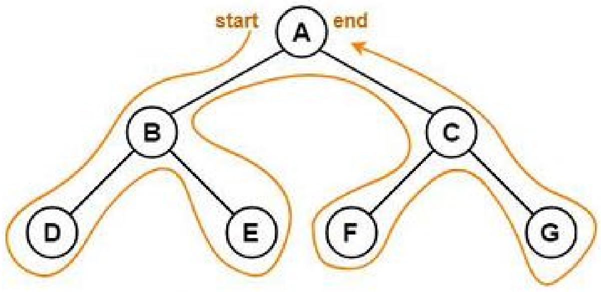
Example-
Consider the following example-



Preorder Traversal : A , B , D , E , C , F , G

Preorder Traversal Shortcut

Traverse the entire tree starting from the root node keeping yourself to the left.



Preorder Traversal : A , B , D , E , C , F , G

Applications-

Preorder traversal is used to get prefix expression of an expression tree.
Preorder traversal is used to create a copy of the tree.

2. Inorder Traversal-

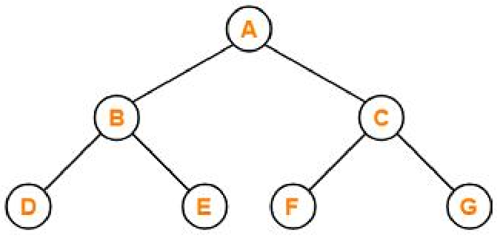
Algorithm-

Traverse the left sub tree i.e. call Inorder (left sub tree)
2. Visit the root
3. Traverse the right sub tree i.e. call Inorder (right sub tree)

Left → Root → Right

Example-

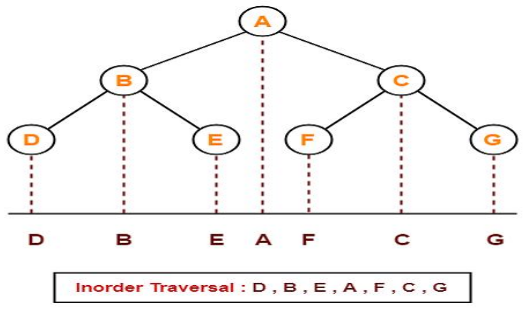
Consider the following example-



Inorder Traversal : D , B , E , A , F , C , G

Inorder Traversal Shortcut

Keep a plane mirror horizontally at the bottom of the tree and
take the projection of all the nodes.



Application-
Inorder traversal is used to get infix expression of an expression tree.

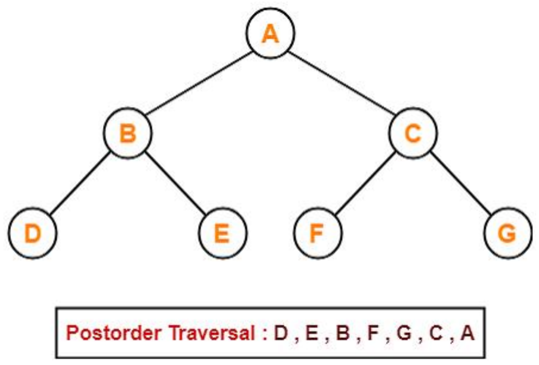
3. Postorder Traversal-

Algorithm-

1. Traverse the left sub tree i.e. call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

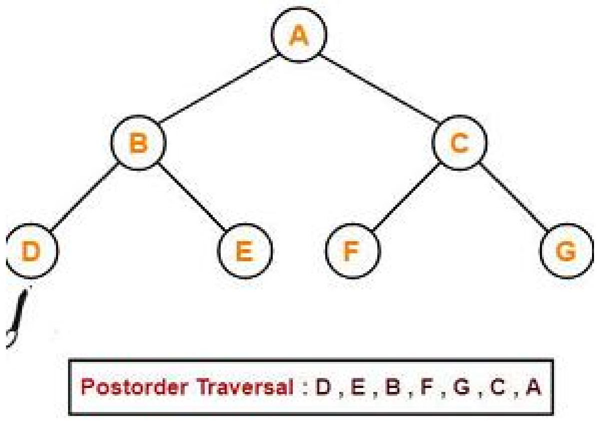
Left → Right → Root

Example-
Consider the following example-



Postorder Traversal Shortcut

Pluck all the leftmost leaf nodes one by one.

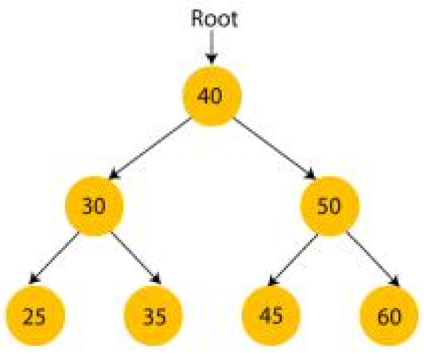


Applications-

Postorder traversal is used to get postfix expression of an expression tree.
Postorder traversal is used to delete the tree.
This is because it deletes the children first and then it deletes the parent.

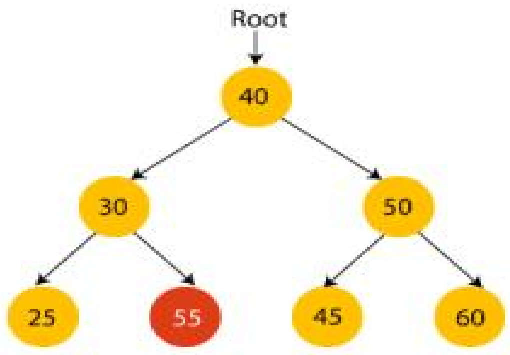
Binary Search tree:

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.
Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

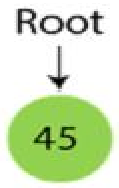
Advantages of Binary search tree
o Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
o As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree
Now, let's see the creation of binary search tree using an example.
Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50
o First, we have to insert 45 into the tree as the root of the tree.

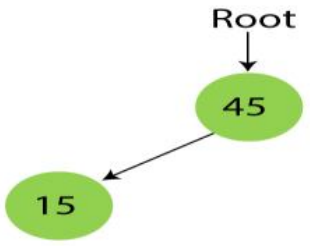
o Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
o Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

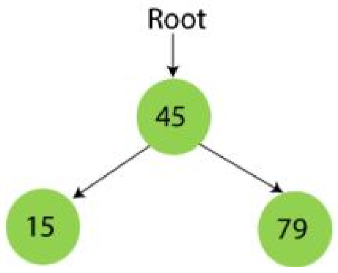
Step 1 - Insert 45.



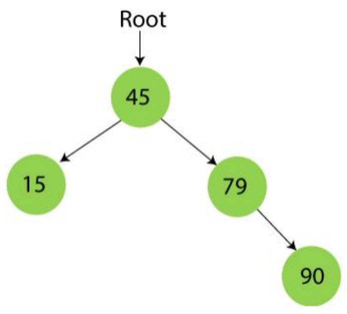
Step 2 - Insert 15.
As 15 is smaller than 45, so insert it as the root node of the left subtree.



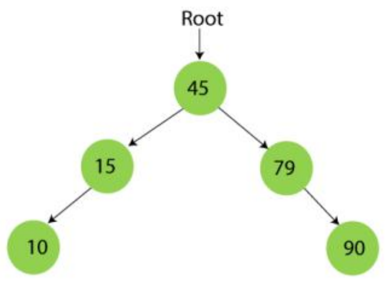
Step 3 - Insert 79.
As 79 is greater than 45, so insert it as the root node of the right subtree.



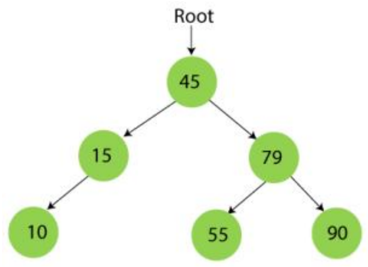
Step 4 - Insert 90.
90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



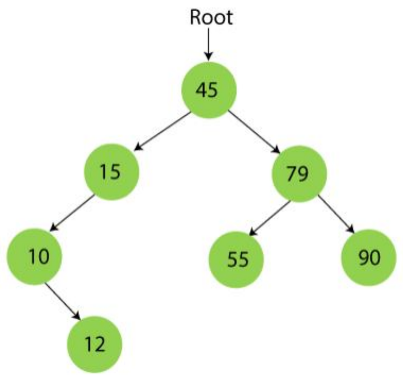
Step 5 - Insert 10.
10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



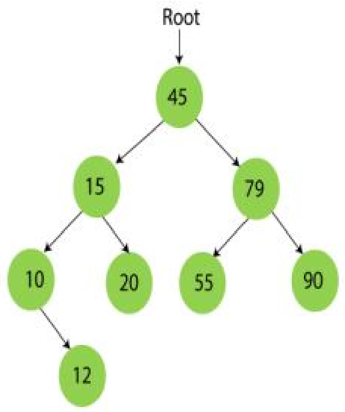
Step 6 - Insert 55.
55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



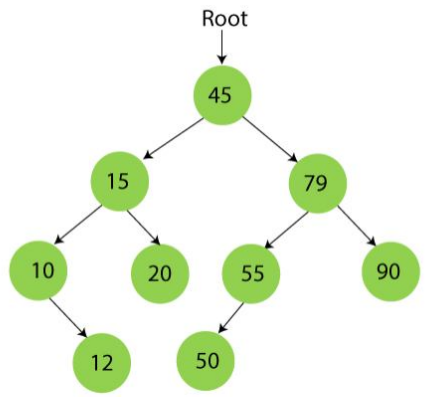
Step 7 - Insert 12.
12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



Step 8 - Insert 20.
20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.
50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

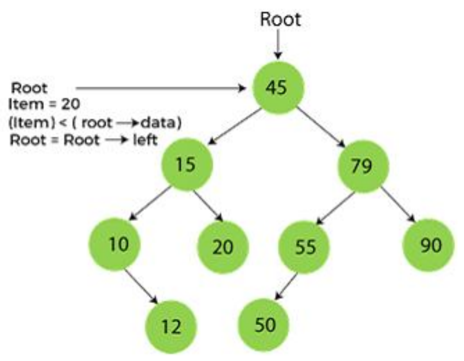
SEARCHING IN BINARY SEARCH TREE :

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

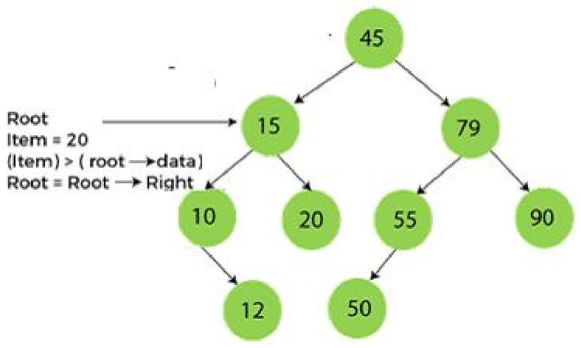
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

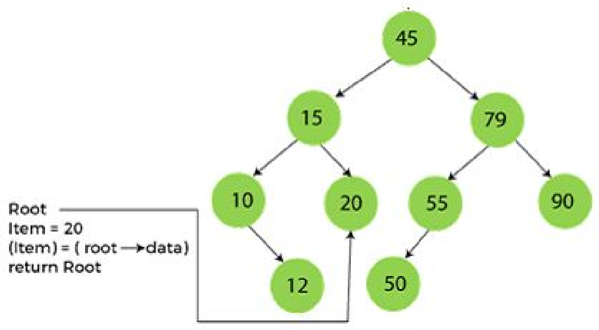
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Algorithm to search an element in Binary search tree

1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root

```
4. else if (item < root → data)
5. return Search(root → left, item)
6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END
```

DELETION IN BINARY SEARCH TREE :

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

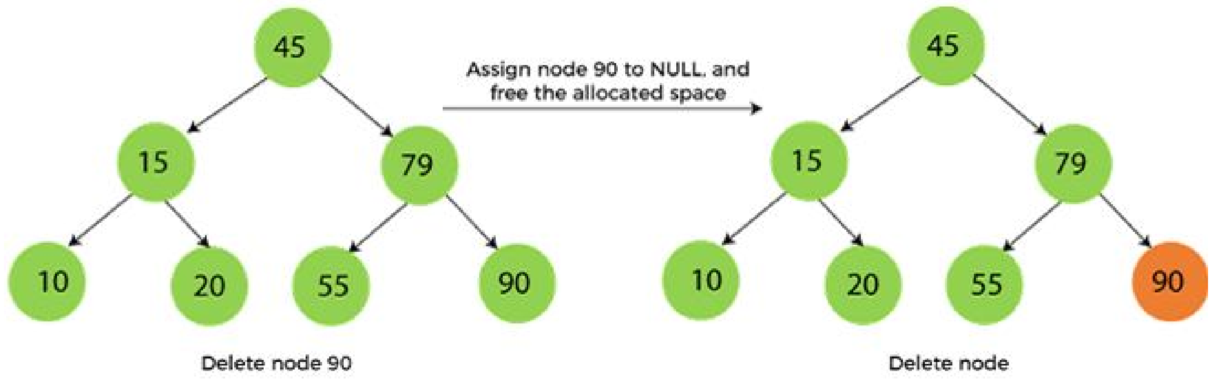
o The node to be deleted is the leaf node, or,
o The node to be deleted has only one child, and,
o The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



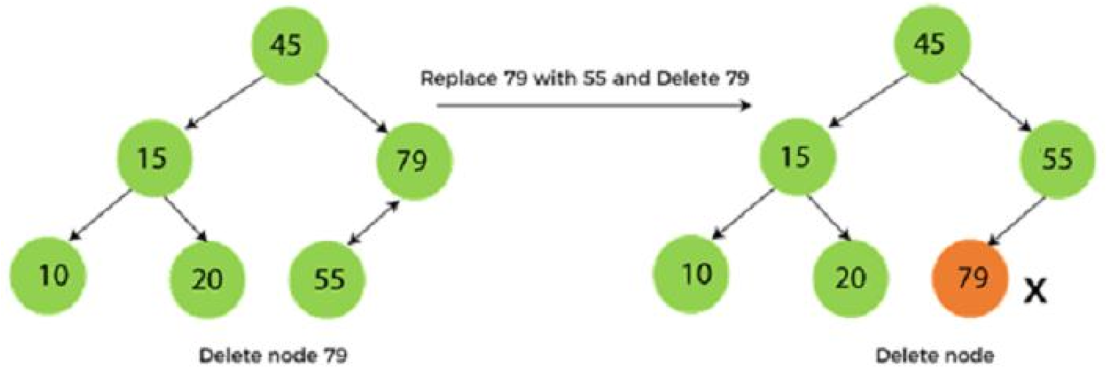
When the node to be deleted has only one child :

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image.

In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



When the node to be deleted has two children:

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

o First, find the inorder successor of the node to be deleted.

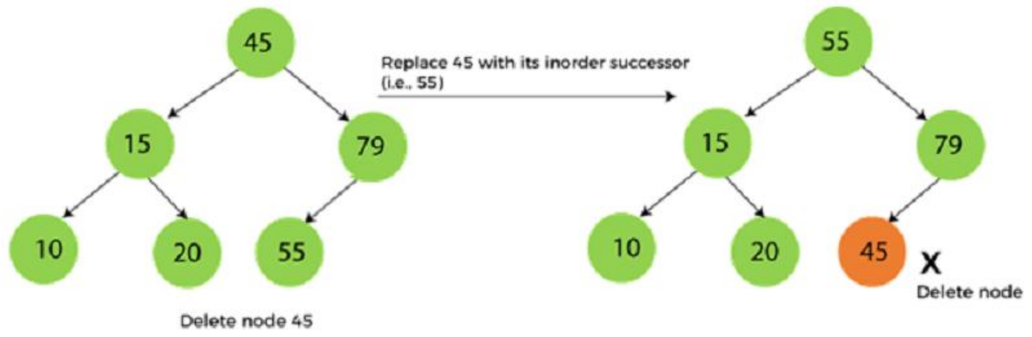
o After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.

o And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor.

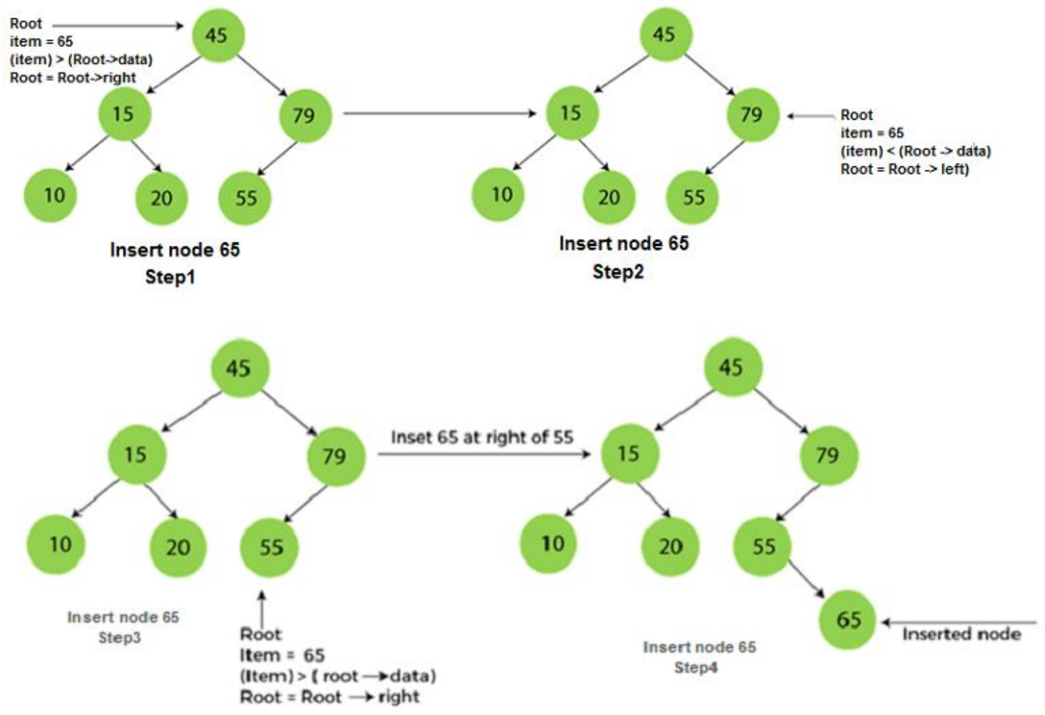
Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



INSERTION IN BINARY SEARCH TREE :

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

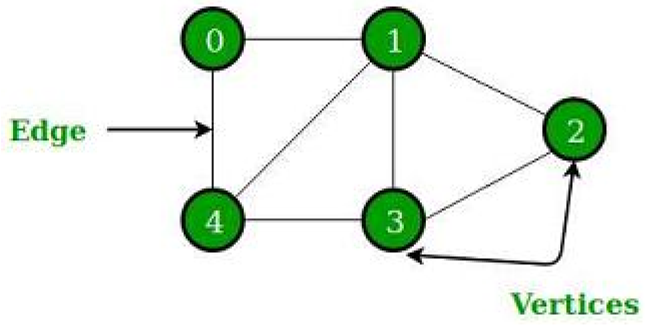
Now, let's see the process of inserting a node into BST using an example.



UNIT-7

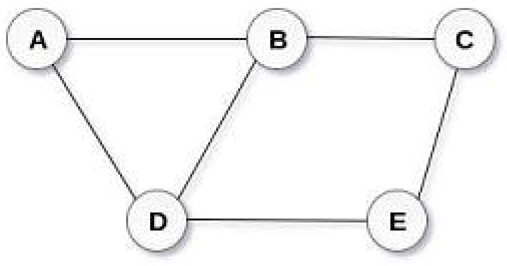
GRAPH

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



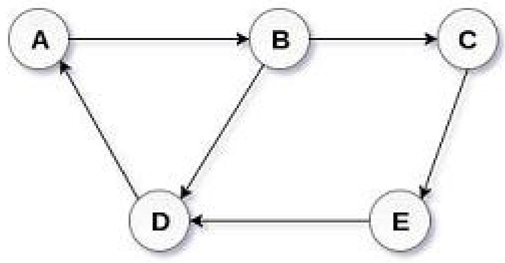
In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.
Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

Definition:
A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.
A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Undirected Graph

DIRECTED AND UNDIRECTED GRAPH
A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.
In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.
A directed graph is shown in the following figure.



Directed Graph

GRAPH TERMINOLOGY :

Path :
A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path :
A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.

Simple Path :
If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.

Cycle :
A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph :
A connected graph is the one in which some path exists between every two vertices (u,v) in V. There are no isolated nodes in connected graph.

Complete Graph :
A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph :
In a weighted graph, each edge is assigned with some data such as length or weight.
The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph :
A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

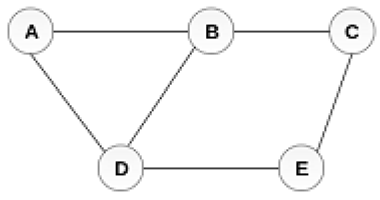
Loop :
An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes :
If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node:
A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph Representation :
By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.
There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.
1. Sequential Representation

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension n x n.
An entry Mij in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between Vi and Vj.
An undirected graph and its adjacency matrix representation is shown in the following figure.

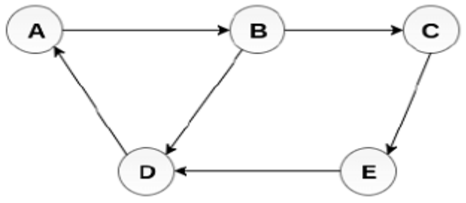


Undirected Graph

$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix

In the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.
There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj.
A directed graph and its adjacency matrix representation is shown in the following figure.

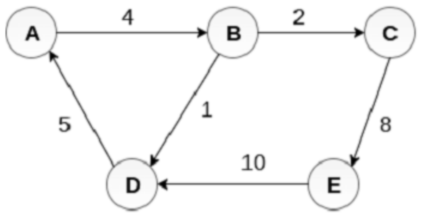


Directed Graph

$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

Adjacency Matrix

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.
The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



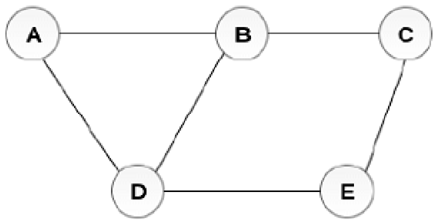
Weighted Directed Graph

$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

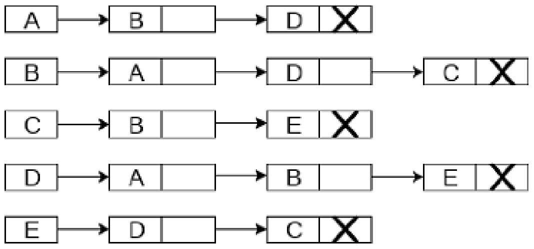
Adjacency Matrix

LINKED REPRESENTATION :
In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.

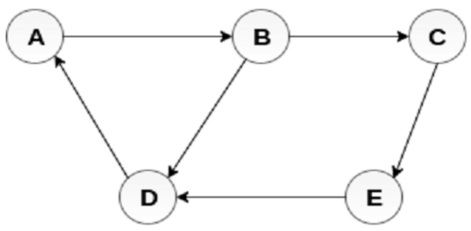


Undirected Graph

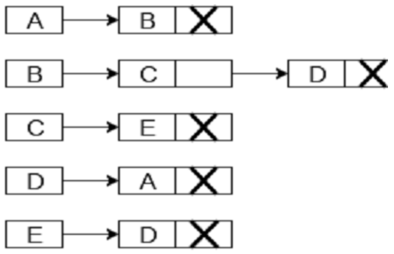


Adjacency List

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.
Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.

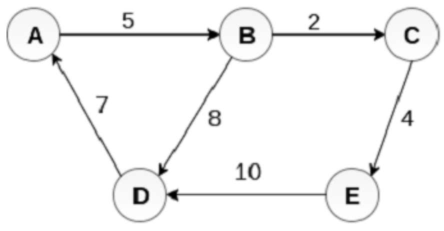


Directed Graph

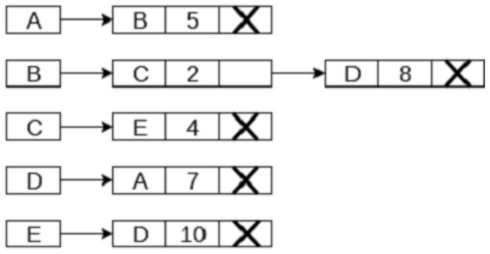


Adjacency List

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.
In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



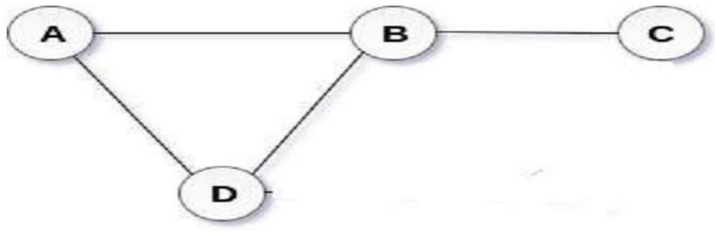
Adjacency List

ADJACENCY MATRIX :
Adjacency matrix definition
In graph theory, an adjacency matrix is a dense way of describing the finite graph structure. It is the 2D matrix that is used to map the association between the graph nodes.
If a graph has n number of vertices, then the adjacency matrix of that graph is n x n, and each entry of the matrix represents the number of edges from one vertex to another.
An adjacency matrix is also called as connection matrix. Sometimes it is also called a Vertex matrix.

Difference between JDK, JRE, and JVM
Adjacency Matrix Representation
If an Undirected Graph G consists of n vertices then the adjacency matrix of a graph is
n x n matrix A = [aij] and defined by -
aij = 1 {if there is a path exists from Vi to Vj}
aij = 0 {Otherwise}
Let's see some of the important points with respect to the adjacency matrix.
- If there exists an edge between vertex Vi and Vj, where i is a row, and j is a column, then the value of aij = 1.
- If there is no edge between vertex Vi and Vj, then the value of aij = 0.
- If there are no self loops in the simple graph, then the vertex matrix (or adjacency matrix) should have 0s in the diagonal.
- An adjacency matrix is symmetric for an undirected graph. It specifies that the value in the ith row and jth column is equal to the value in jth row ith column.

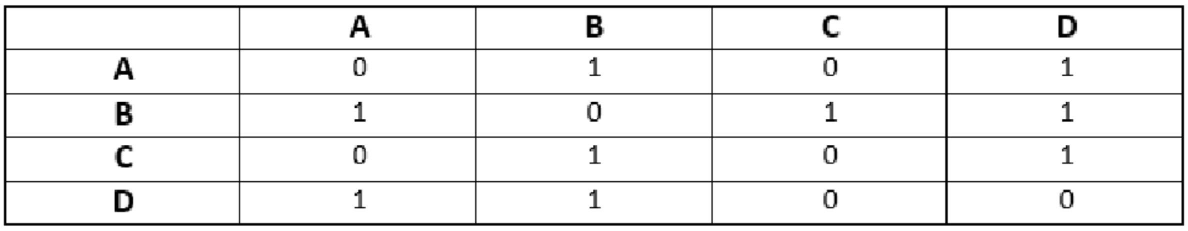
- If the adjacency matrix is multiplied by itself, and if there is a non-zero value is present at the ith row and jth column, then there is the route from Vi to Vj with a length equivalent to 2. The non-zero value in the adjacency matrix represents that the number of distinct paths is present.

Note: In an adjacency matrix, 0 represents that there is no association is exists between two nodes, whereas 1 represents that there is an association is exists between two nodes.



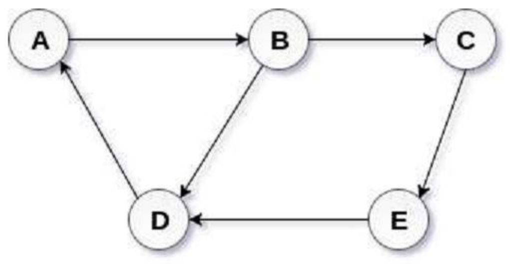
Undirected Graph

In the graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix will be 0. The adjacency matrix of the above graph will be -



	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	0	0

ADJACENCY MATRIX FOR A DIRECTED GRAPH :
In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called the initial node, while node B is called the terminal node.
Let us consider the below directed graph and try to construct the adjacency matrix of it.



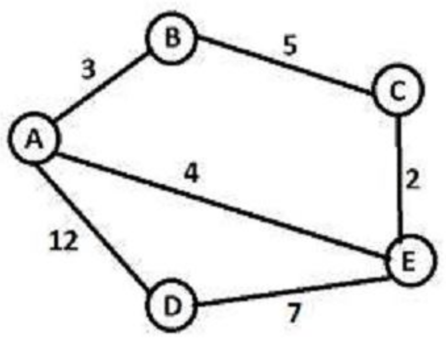
Directed Graph

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix will be 0. The adjacency matrix of the above graph will be -

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

NOTE: A graph is said to be the weighted graph if each edge is assigned a positive number, which is called the weight of the edge.

Question 1 - What will be the adjacency matrix for the below undirected weighted graph?

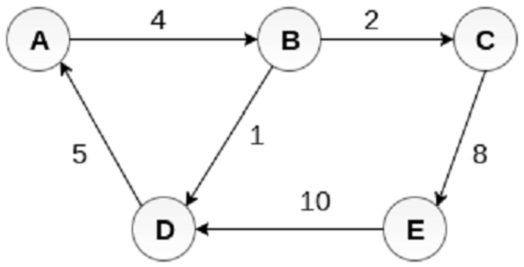


Solution - In the given question, there is no self-loop, so it is clear that the diagonal entries of the adjacent matrix for the above graph will be 0. The above graph is a weighted undirected graph. The weights on the graph edges will be represented as the entries of the adjacency matrix.

The adjacency matrix of the above graph will be –

	A	B	C	D	E
A	0	3	0	12	4
B	3	0	5	0	0
C	0	5	0	0	2
D	12	0	0	0	7
E	4	0	2	7	0

Question 2 - What will be the adjacency matrix for the below directed weighted graph?



Solution - In the given question, there is no self-loop, so it is clear that the diagonal entries of the adjacent matrix for the above graph will be 0. The above graph is a weighted directed graph. The weights on the graph edges will be represented as the entries of the adjacency matrix.

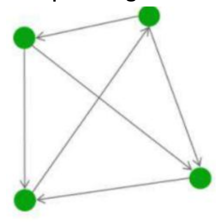
The adjacency matrix of the above graph will be -

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Hope this article is beneficial to you in order to understand about adjacency matrix. Here, we have discussed the adjacency matrix along with its creation and properties. We have also discussed the formation of adjacency matrix on directed or undirected graphs, whether they are weighted or not.

PATH MATRIX:

A path matrix is a matrix representing a graph where each value in m'th row and n'th column project whether there is a path from m to n. The path may be direct or indirect. It may have a single edge or multiple edges.



$$P = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Path Matrix for a Graph

Consider the graph:

Adjacency Matrix for this graph is:

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Derive A2 : Derive A2 :

$$A^2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Derive A3 :

$$A^3 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

Derive A4 :

$$A^4 = \begin{pmatrix} 1 & 0 & 2 & 2 \\ 2 & 1 & 2 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 2 & 0 & 1 \end{pmatrix}$$

Sum up all four matrix to derive B4 :

$$B_4 = \begin{pmatrix} 2 & 2 & 4 & 4 \\ 3 & 3 & 4 & 3 \\ 1 & 1 & 2 & 3 \\ 1 & 3 & 2 & 3 \end{pmatrix}$$

Derive Path Matrix P from B4 by replacing any none zero value with 1:

$$\mathrm{P} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Difference Between Adjacency Matrix & Path Matrix :

Key difference between Adjacency Matrix and Path Matrix is that an adjacency matrix is about direct edge where a path matrix is about whether can be traveled or not. Path matrix include both direct and indirect edges.

UNIT- 8

SORTING SEARCHING & MERGING

SEARCHING :

It is the process of finding an element from a list of elements. Search operation also provides the location of the element if it is found otherwise the operation is unsuccessful. Generally there are two types of searching methods:

1. Linear Search
2. Binary Search

1. Linear Search:

It is the simplest of searching methods. In this method the item is compared with each element of the list. If the item= element then the item is said to be found, otherwise the item is not found. The time complexity of the linear search is O(n). This is linear time complexity and is suitable for small list of items. But when the list of elements is very huge then the search time becomes very huge in worst case.

Algorithm:

```
LINEAR (DATA, N, ITEM, LOC)
Step 1: [Insert ITEM at the end of data]
            Set DATA [N+1] = ITEM
Step 2: [Initialize counter]
           Set LOC=1
Step 3: [Search for ITEM]
            Repeat while DATA [LOC]!= ITEM
Step 4: [Successful]
           If LOC=N+1
          Then Set LOC = 0
Step 5: Exit
```

BINARY SEARCH

Suppose DATA is an array which is sorted in increasing numerical order or equivalently, alphabetically. Then there is an extremely efficient searching algorithm, called binary search.

Algorithm

```
Binary search (DATA, LB, UB, ITEM, LOC)
Step 1: [Initialize the segment variables]
           Set BEG := LB, END := UB and MID := INT ((BEG + END)/2)
Step 2: [Loop]
           Repeat Step 3 and Step 4 while BEG <= END and DATA [MID] != ITEM
Step 3: [Compare]
           If ITEM < DATA [MID]
           then set END := MID - 1
Else
            Set BEG = MID + 1
Step 4: [Calculate MID]
           Set MID := INT ((BEG + END)/2)
Step 5: [Successful search]
           If DATA [MID] = ITEM
           then set LOC := MID
           Else set LOC := NULL
Step 6: Exit
```

Complexity of the Binary Search Algorithm

The complexity is measured by the number f(n) of comparison to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most f(n) comparison to locate ITEM where

$2f(n) > n$

Or equivalently

$F(n) = [\log 2\ n] + 1$

The running time for the worst case is approximately equal to log2 n and the average case is approximately equal to the running time for the worst case.

SORTING:

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing with numerical data or alphabetically with character data.

BUBBLE SORT:

The bubble sort has no reading characteristics. It is very slow, no matter what data it is sorting. This algorithm is included for the shake of completeness not because of any merit. As the largest element is bubble of sinks up to its final position. It is known as bubble sort.

Algorithm:

BUBBLE (DATA,N)

Here DATA is an array with N element. This algorithm sorts the element in DATA.

```
Step 1: [Loop]
             Repeat step 2 and step 3 for K=1 to N-1
Step 2: [Initialize pass pointer PTR]
            Set[PTR]=1
Step 3: [Execute pass]
            Repeat while PTR <=N-K
a.  If DATA [PTR] > DATA [PTR+1]
             Then interchange DATA [PTR] & DATA [PTR+1]
              [End of if structure]
b.  Set PTR =PTR+1
              [End of Step 1 Loop]
Step 4: Exit
```

Complexity of the Bubble Sort Algorithm

The time for a sorting algorithm is measured in terms of the number of comparisons. The number f(n) of comparisons in the bubble sort is easily computed. There are n-1 comparisons during the first pass, which places the largest element in the last position; there are n-2 comparisons in the second step, which places the second largest element in the next to last position and so on.

F(n)=(n-1)+(n-2)+….+2+1=n(n-1)/2=n^2/2+0(n)=0(n^2)

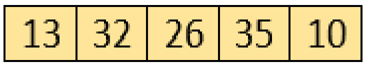
The time required to execute the bubble sort algorithm is proportional to n^2, Where n is the number of input items.

Working of Bubble sort Algorithm:

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

To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is O(n2).

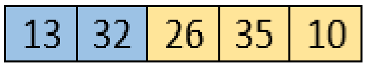
Let the elements of array are –



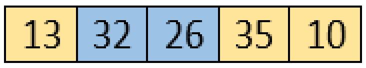
13	32	26	35	10

First Pass

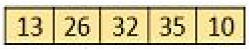
Sorting will start from the initial two elements. Let compare them to check which is greater.



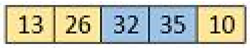
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



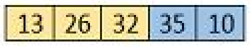
Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -



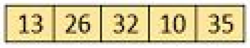
Now, compare 32 and 35.



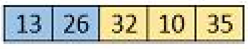
Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.
Now, the comparison will be in between 35 and 10.

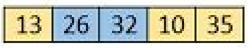


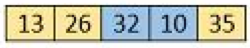
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -



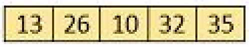
Now, move to the second iteration.
Second Pass
The same process will be followed for second iteration.

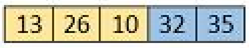




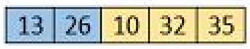


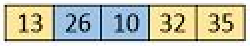
Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -



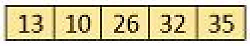


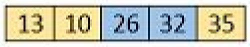
Now, move to the third iteration.
Third Pass
The same process will be followed for third iteration.

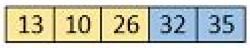




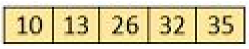
Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -







Now, move to the fourth iteration.
Fourth pass
Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

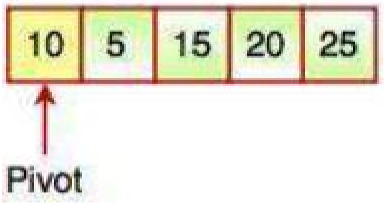
Bubble sort complexity :

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

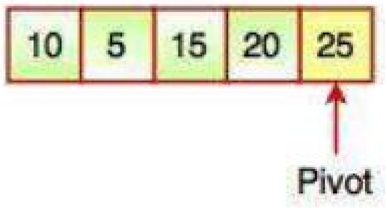
Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is O(n).
Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is O(n^2).
Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is O(n^2).

QUICK SORT:
Quick sort is also known as Partition-exchange sort based on the rule of Divide and Conquer.
It is a highly efficient sorting algorithm.
Quick sort is the quickest comparison-based sorting algorithm.
It is very fast and requires less additional space
Quick sort picks an element as pivot and partitions the array around the picked pivot.
There are different versions of quick sort which choose the pivot in different ways:

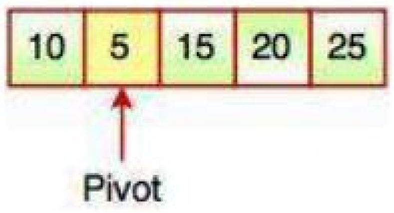
1. First element as pivot



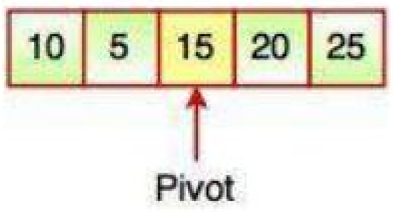
2. Last element as pivot



3. Random element as pivot



4. Median as pivot



Example of Quick Sort:
1. 44 33 11 55 77 90 40 60 99 22 88
Let 44 be the Pivot element and scanning done from right to left
Comparing 44 to the right-side elements, and if right-side elements are smaller than 44, then swap it. As 22 is smaller than 44 so swap them.

22 33 11 55 77 90 40 60 99 44 88

Now comparing 44 to the left side element and the element must be greater than 44 then swap them. As 55 are greater than 44 so swap them.

22 33 11 44 77 90 40 60 99 55 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element 44 & one right from pivot element.

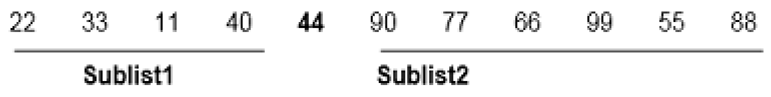
22 33 11 40 77 90 44 60 99 55 88

Swap with 77:

22 33 11 40 44 90 77 60 99 55 88

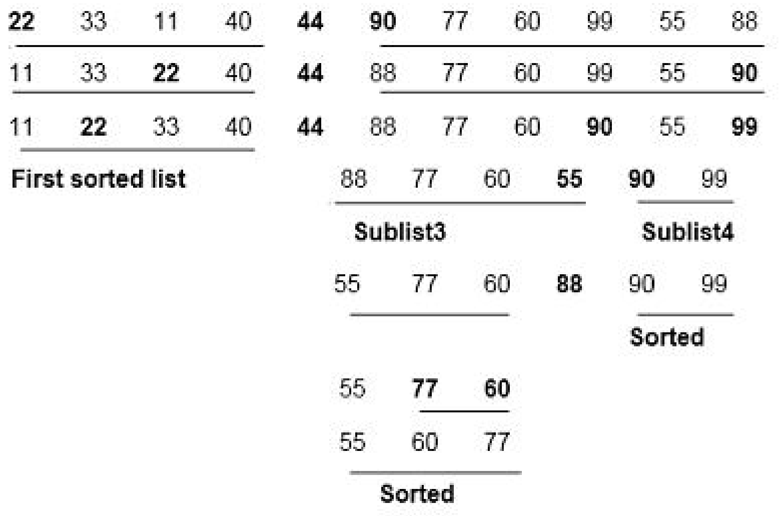
Now, the element on the right side and left side are greater than and smaller than 44 respectively.

Now we get two sorted lists:

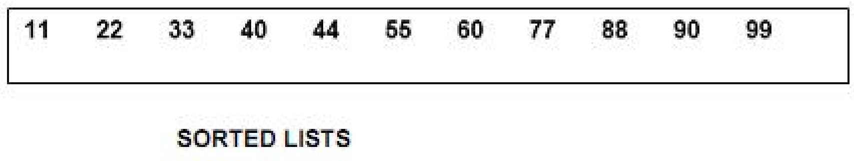


And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.



Merging Sublists:



Algorithm:

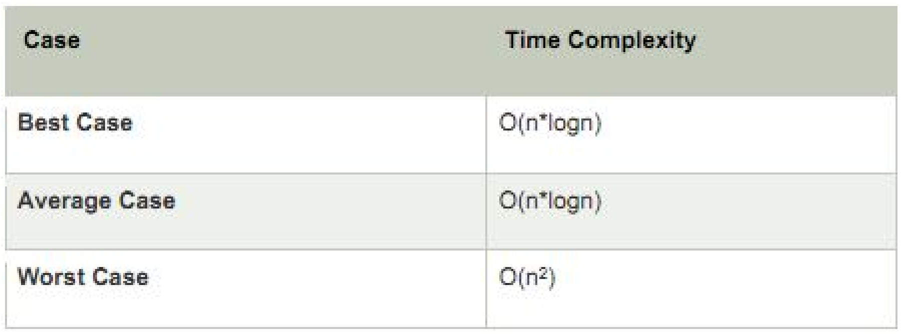
```
QUICKSORT (array A, start, end)
{
 1 if (start < end)
 2 {
3 p = partition(A, start, end)
4 QUICKSORT (A, start, p - 1)
5 QUICKSORT (A, p + 1, end)
6 }
}
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

```
PARTITION (array A, start, end)
 {
   pivot = A[end]
   i = start-1
   for j = start to end -1 {
   do if (A[j] < pivot) {
   then i = i + 1
   swap A[i] with A[j]
    }}
   swap A[i+1] with A[end]
   return i+1
}
```

Complexity of the Quick Sort Algorithm



Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n^2)

MERGING:

The operation of sorting is closely related to the process of merging. The merging of two order table which can be combined to produce a single sorted table. This process can be accomplishes easily by successively selecting the record with the smallest key occurring by either of the table and placing this record in a new table.

SIMPLE MERGE

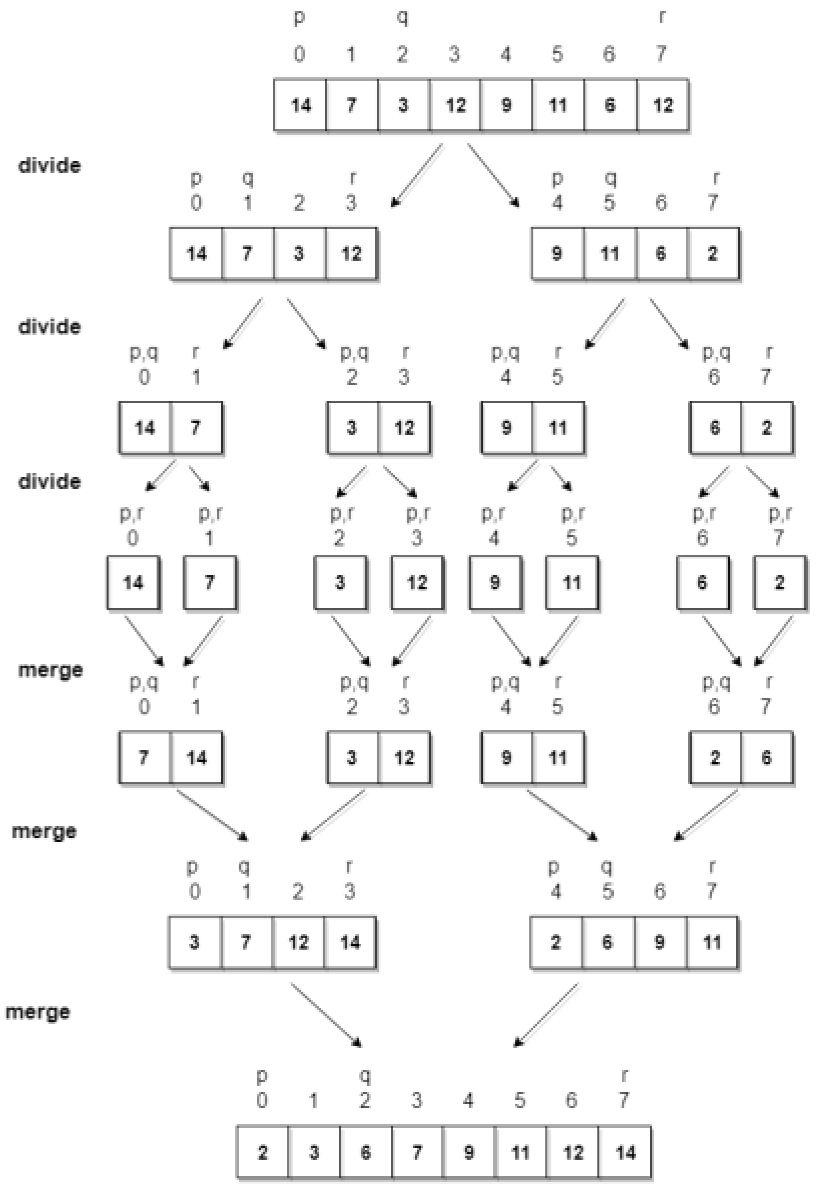
SIMPLE MERGE [FIRST,SECOND,THIRD,K]

Given two orders in table sorted in a vector K with FIRST, SECOND, THIRD. The variable I & J denotes the curser associated with the FIRST & SECOND table respectively. L is the index variable associated with the vector TEMP.

```
Algorithm
Step 1: [Initialize]
            Set I = FIRST
            Set J = SECOND
            Set L = 0
Step 2: [Compare corresponding elements and output the smallest]
             Repeat while I < SECOND & J < THIRD
             If K[I] <= K[J],then L=L+1
            TEMP [L]=K[I]
             I=I+1
             Else L=L+1
             TEMP[L]=K[J]
             J=J+1
Step 3: [Copy remaining unprocessed element in output area]
             If I>=SECOND
            Then repeat while J<=THIRD
            L=L+1
            TEMP[L]=K[J]
            J=J+1
```

```
Else
        Repeat while I< SECOND
        L=L+1
        TEMP[L]=K[I]
        I=I+1
Step 4: [Copy the element into temporary vector into original area]
        Repeat for I=1,2,…..L
        K[FIRST-I+1]=TEMP[I]
Step 5: Exit
```

Working of Merge Sort:



Complexity of the Merging Algorithm

The input consists of the total number n=r+s of elements in A and B. Each comparison assigns an elements to the array C, which eventually has n elements. Accordingly, the number f(n) of comparisons cannot exceed n:

F(n) <= n = 0(n)

In other words, the merging algorithm can be run in linear time.

WORST CASE : n log= 0[n log n]

AVERAGE CASE : n log n=0[n log n]

REVIEW QUESTIONS

1. What is the principle of bubble sort?
2. Differentiate between linear search and binary search?
3. What is searching and what are the methods of searching?

5 Marks

1. Explain quick sort and write the algorithm.
2. Write an algorithm for binary search.
3. Write an algorithm to implement bubble sort and mention its time complexity.

7 MARKS

1. Write an algorithm for merging and mention its complexity.
2. What is linear search? Write an algorithm and mention its complexity.