Data Structure Notes

STUDENT ADVISORY

Dear Students,

Please be informed that the notes provided by the institute offer a concise presentation of the syllabus. While these notes are helpful for an overview and quick revision, We would strongly suggest that you refer to the prescribed textbooks / Reference book for a comprehensive understanding and thorough preparation of all exams and writing in the examination.

Best regards,

LJ Polytechnic.

પ્રિય વિદ્યાર્થીઓ,

તમને જાણ કરવામા આવે છે કે સંસ્થા દ્વારા પ્રદાન કરવામાં આવેલી નોંધો અભ્યાસક્રમની સંક્ષિપ્ત પ્રસ્તુતિ આપે છે. આ નોંધો વિહંગાવલોકન અને ઝડપી પુનરાવર્તન માટે મદદરૂપ હોઈ શકે છે તેમ છતા, અમે ભારપૂર્વક સૂચન કરીએ છીએ કે વિદ્યાર્થી તમામ પરીક્ષાઓ અને પરીક્ષામાં લેખનની વ્યાપક સમજણ અને સંપૂર્ણ તૈયારી માટે માત્ર સૂચવેલા પાઠ્યપુસ્તકો/સંદર્ભ પુસ્તકનો સંદર્ભ લો.

એલજે પોલિટેકનિક.

Introduction to Data Structures

Q.1.Define the terms associated with Data structure and explain classification in detail.

Data Structure

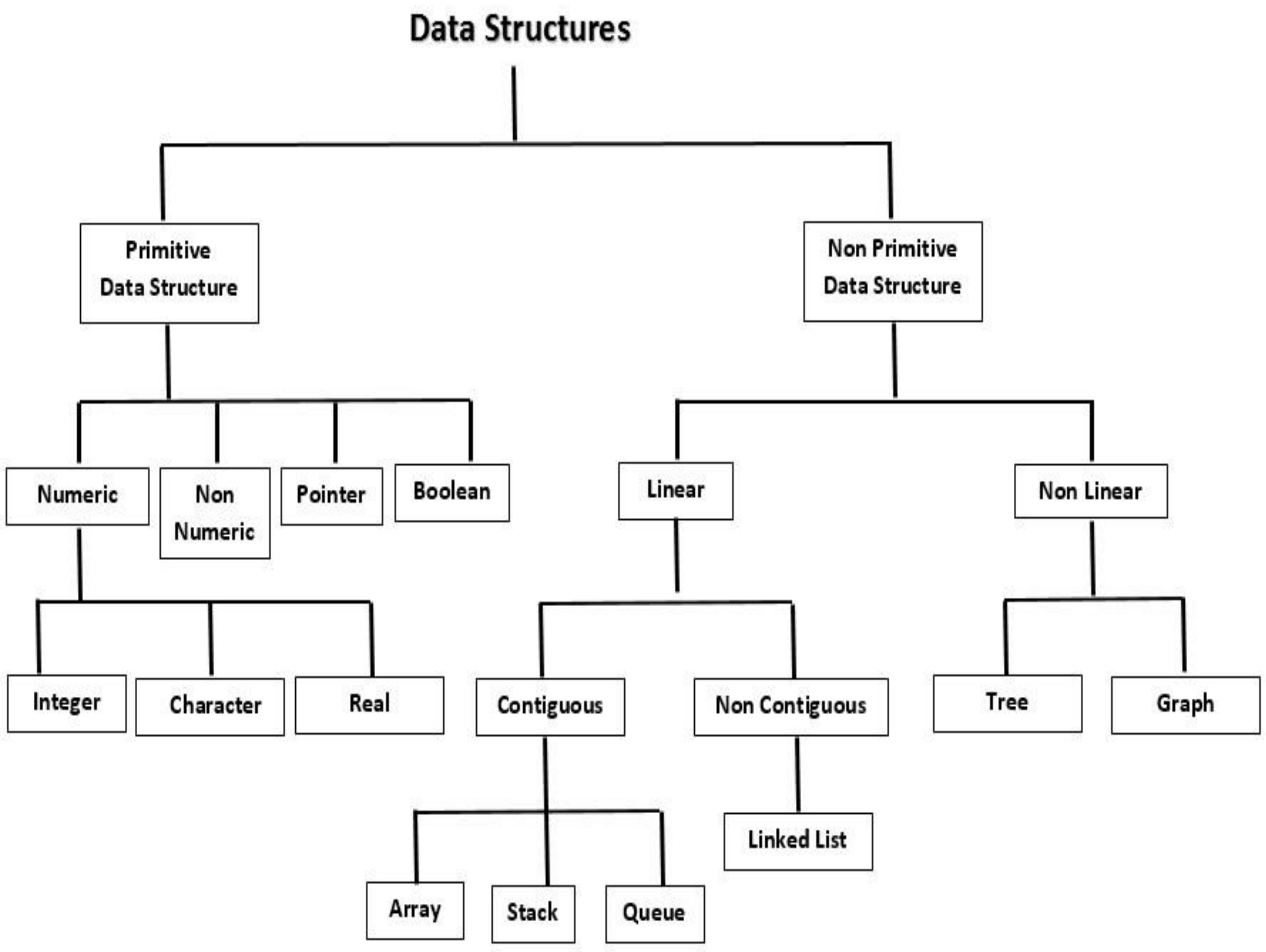
Definition

Data structure is a representation of logical relationship existing between individual elements of data.

Introduction

A data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

Classification of Data Structure



Primitive Data structure

Primitive Data Structures are the basic data structures that directly operated by machine level instructions. They have different memory representations on different computers. All primary data types are known as primitive data structures. Examples: Integers, floating point numbers, character constants, and pointers come under this category.

Non - Primitive Data structure

Non primitive data structures are derived data structure and does not directly operate by machine level instructions. Non-primitive data structures are more complicated data structures than primitive data structures. Non primitive data structures are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Examples: Arrays, lists and files come under this category.

Linear Data structure

A Linear data structure have data elements arranged in sequential manner and each member element is connected to its previous and next element. This connection helps to traverse a linear data structure in a single level and in single run. Such data structures are easy to implement as computer memory is also sequential. Examples of linear data structures are List, Queue, Stack, Array etc.

Non - Linear Data structure

A non-linear data structure has no set sequence of connecting all its elements and each element can have multiple paths to connect to other elements. Such data structures support multi-level storage and often cannot be traversed in single run. Such data structures are not easy to implement but are more efficient in utilizing computer memory. Examples of non-linear data structures are Tree, Graphs etc.

Contiguous Data structure

Contiguous structures can be broken drawn further into two kinds: Those that contain data items of all the same size, and those where the size may differ. The first kind is called the array. In an array, each element is of the same type, and thus has the same size. The second kind of contiguous structure is called structure, in a structure, elements may be of different data types and thus may have different sizes.

Non - Contiguous Data structure

Non-contiguous structures are implemented as a collection of data-items, called nodes, where each node can point to one or more other nodes in the collection. The simplest kind of noncontiguous structure is linked list. A linked list represents a linear, one-dimension type of non-contiguous structure, where there is only the notation of backwards and forwards.

A tree is an example of a two-dimensional non-contiguous structure. Here, there is the notion of up and down and left and right. In a tree each node has only one link that leads into the node and links can only go down the tree. The most general type of non-contiguous structure, called a graph has no such restrictions.

Homogeneous Data structure

Homogeneous data structures are ones that can store a single type of data (numeric, integer, character etc.) Example: Array.

Non - Homogeneous Data structure

Non - Homogeneous data structure are those structures that contains a variety or dissimilar type of data. Example: Structure.

Q.2. Explain advantages of data structure.

Advantages of data structure

1) Data structures allow information storage on hard disks.
2) Provides means for management of large dataset such as databases or internet indexing services.
3) Are necessary for design of efficient algorithms.
4) Allows safe storage of information on a computer. The information is then available for later use and can be used by multiple programs. Additionally, the information is securing and cannot be lost (especially if it is stored on magnetic tapes).
5) Allows the data use and processing on a software system.
6) Allows easier processing of data.
7) Using internet, we can access the data anytime from any connected machine (computer, laptop, tablet, phone etc.)

Q.3. Explain applications of data structure.

Applications of data structure

The data structures store the data according to the mathematical or logical model it is based on. The type of operations on a certain data structure makes it useful for specific tasks. Here is a brief discussion of different applications of various data structures.

Arrays

1) Storing list of data elements belonging to same data type.
2) Auxiliary storage for other data structures.
3) Storage of binary tree elements of fixed count.
4) Storage of matrices.

Linked List

1) Implementing stacks, queues, binary trees and graphs of predefined size.
2) Implement dynamic memory management functions of operating system.
3) Polynomial implementation for mathematical operations
4) Circular linked list is used to implement OS or application functions that require round robin execution of tasks.
5) Doubly linked list is used in the implementation of forward and backward buttons in a browser to move backwards and forward in the opened pages of a website.
6) Circular queue is used to maintain the playing sequence of multiple players in a game.

Stack

1) Temporary storage structure for recursive operations.
2) Evaluation of arithmetic expressions in various programming languages.
3) Conversion of infix expressions into postfix expressions.
4) Checking syntax of expressions in a programming environment.

String

1) In all the problems solutions based on backtracking.
2) Used in depth first search in graph and tree traversal.
3) UNDO and REDO functions in an editor.

Queues

1) It is used in breadth search operation in graphs.
2) Job scheduler operations of OS like a print buffer queue, keyboard buffer queue to store the keys pressed by users.
3) Job scheduling, CPU scheduling, Disk Scheduling.
4) Priority queues are used in file downloading operations in a browser.

Trees

1) Implementing the hierarchical structures in computer systems like directory and file system.
2) Implementing the navigation structure of a website.
3) Decision making in gaming applications.
4) Implementation of priority queues for priority-based OS scheduling functions.
5) Parsing of expressions and statements in programming language compilers.

Q.4. Define algorithm. Write advantages of using algorithms.

Algorithm

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. In other words, an algorithm is sequence of finite steps to complete a task. An algorithm is a procedure for solving a problem. Algorithms are generally created independent of underlying languages, i.e., an algorithm can be implemented in more than one programming language.

Properties of an Algorithm

Key Features

Input - An algorithm should have 0 or more finite number of well-defined inputs.

Output - An algorithm should have 1 or more well-defined outputs, and should match the desired output.

Definiteness - Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs and outputs should be clear and must lead to only one meaning.

Finiteness - Algorithms must terminate after a finite number of steps.

Effectiveness – Every step of an algorithm must be effective or have some meaning.

Feasibility - An algorithm must be feasible. It means it should be possible to implement.

Correctness - For every input an algorithm should produce correct output.

Example of an algorithm

Write an algorithm to find average of two number.

Algorithm for AVERAGE of two number

```
Step 1:  Take two inputs A and B
Step 2:  Calculate sum = A+B
Step 3:  Calculate Average = sum/2
Step 4:  Print “Average”
Step 5:  Finish
```

Q.5. Define terms: (A) Time complexity (B) Space complexity.

Time Complexity

Time Complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input.

Space Complexity

It is the total memory space required by the program for its execution.

Q.6.Explain types of Time complexity OR Explain kinds of algorithm analysis.

Best Case Time Complexity

1) The minimum amount of the time taken by an algorithm on input size n is called best case time complexity.
2) It is denoted by the symbol Big Omega (Ω).
3) It executes in minimum number of steps.
4) Example: The best case time complexity of Quick sort is Ω (nlogn).

Average Case Time Complexity

1) The average amount of the time taken by an algorithm on input size n is called best case time complexity.
2) It is denoted by the symbol Big Theta (Ɵ).
3) It executes in average number of steps.
4) Example: The average case time complexity of Quick sort is Ɵ (nlogn).

Worst Case Time Complexity

1) The maximum amount of the time taken by an algorithm on input size n is called best case time complexity.
2) It is denoted by the symbol Big Oh (O).
3) It executes in maximum number of steps.
4) Example: The worst case time complexity of Quick sort is O (n2).

Q.7. Explain characteristics of Array.

1) An array holds elements that have the same data type.
2) Array elements are stored in subsequent memory locations.
3) Two-dimensional array elements are stored row by row in subsequent memory locations.
4) Array name represents the address of the starting element.
5) Array size should be mentioned in the declaration. Array size must be a constant expression and not a variable.

Q.8. Explain One-dimensional Array.

Array

One Dimensional Array

An array is defined as an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations in RAM. The elements of an array are of same data type and each item can be accessed using the same name.

Declaration of an array: We know that all the variables are declared before they are used in the program. During declaration, the size of the array has to be specified. The size used during declaration of the array informs the compiler to allocate and reserve the specified memory locations.

Syntax: data_type array_name[n];
Where, n is the number of data items (or) index (or) dimension. 0 to (n-1) is the range of array.

Example: int a[5];
float x[10];

Initialization of Arrays: We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The two different types of initializing arrays are:
1. At Compile time
2. At Run Time

Compile Time Initialization:

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is

Data_type array_name[size] = {list of values};

(i) Initializing all specified memory locations:

Arrays can be initialized at the time of declaration when their initial values are known in advance. Array elements can be initialized with data items of type int, char etc.

Example: int a[5]={10,15,1,3,20};

(ii) Partial array initialization:

Partial array initialization is possible in c language. If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zero automatically.

Example: int a[5]={10,15};

Initialization with all zeros:

Example: int a[5]={0};

Run Time Initialization:

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. scanf can be used to initialize an array.
Example:
```
int x[3];
scanf("%d %d %d", &x[0],&x[1],&x[2]);
```

The above statements will initialize array elements with the values entered through the key board. We can also use for loop like,

```
for(i=0;i<3;i++)
scanf("%d",&x[i]);
```

Display elements of array:

We can display elements of array as,

```
printf("%d %d %d",x[0],x[1],x[2]);
```

OR

We can also use for loop like

```
for(i=0;i<3;i++)
printf("%d",x[i]);
```

Q.9. Explain Two-Dimensional Array.

Two-Dimensional Array

An array consisting of two subscripts is known as two-dimensional array. In two dimensional arrays the array is divided into rows and columns.

Declaration

Syntax: data_type array_name[row_size][column_size];

Example: int arr[3][3];

Where first index value shows the number of the rows and second index value shows the number of the columns in the array.

Initializing two-dimensional arrays:

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Ex: int a[2][3]={0,0,0,1,1,1};

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row.

The above statement can also be written as,

```
int a[2][3] = {{ 0,0,0},{1,1,1}};
```

by surrounding the elements of each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below

```
int a[2][3]={
{0,0,0},
{1,1,1}
};
```

When the array is completely initialized with all values, explicitly we need not specify the size of the first dimension.

Example : int a[][3] = { {0,2,3}, {2,1,2} };

If the values are missing in an initializer, they are automatically set to zero.

Example: int a[2][3] = { {1,1}, {2} };
Will initialize the first two elements of the first row to one, the first element of the second row to two and all other elements to zero.

Run Time Initialization :

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. scanf can be used to initialize an array.
Example:

```
int  x[2][3];
scanf("%d %d %d %d %d %d",&x[0][0],&x[0][1],&x[0][2] ,&x[1][0],&x[1][1],&x[1][2]);
```

The above statements will initialize array elements with the values entered through the key board.

OR
We can also use for loop like,

```
for(i=0;i<2;i++)
{
        for(j=0;j<3;j++)
        {
        scanf("%d",  &x[i][j]);
        }
}
```

Display elements of array:

We can display elements of array as

```
printf("%d%d%d%d%d%d",  x[0][0],x[0][1],x[0][2] ,x[1][0],x[1][1],x[1][2]);
```

OR

We can also use for loop like

```
for(i=0;i<2;i++)
{
        for(j=0;j<3;j++)
        {
        printf("%d",x[i][j]);
        }
}
```

Q.10. Define String. Explain how to declare and initialize string.

Strings

Definition

In C language a string is group of characters (or) array of characters, which is terminated by delimiter \0 (null).

Declaring Strings

C does not support string as a data type. It allows us to represent strings as character arrays. In C, a string variable is any valid C variable name and is always declared as an array of characters.

Syntax: char string_name[size];
The size determines the number of characters in the string name.
Example: char city[10];
char name[30];

Initializing strings
There are several methods to initialize values for string variables.
Example: char city[] = ”NEWYORK”;
char city[8] ={‘N’,’E’,’W,’Y’,’O’,’R’,’K,’\0’};
The string city size is 8 but it contains 7 characters and one character space is for NULL terminator.

Q.11. Explain read and write function of string.

String read functions

scanf()
The library function scanf() is most popular standard input function used to read string from the keyboard.

getchar()
This library function is used to get single character from the terminal.
char ch;
ch = getchar();

gets()
This library function read line of text containing white space until new line character.
char str[5];
gets(str);

String write functions

printf()
The standard printf function is used for printing or displaying a string on an output device. The format specifier used is %s.
printf("%s", name);

puts()
The puts function prints the string on an output device and moves the cursor back to the first position.
char str[5]=”data”;
puts(str);

putchar()
The C library function putchar() is used to print a single character on the terminal.
char str[5]=”data”;
int i;
for(i=0;i<10;i++)
putchar(str[i]);

Q.12. Explain various string operations.

String Operations

Operations performed on String

Followings are some common operations performed on strings

1. Find length of given string.
2. Compare two strings.
3. Copy one string into other string.
4. Concatenate two strings.

Algorithm for STRING LENGTH operation

```
STR_LEN (str)
str: Given string
Step 1: [Initialization]
            length = 0
Step 2: [Process until end of the string]
            repeat while (str [length] ≠ NULL)
            length = length + 1
Step 3: [Finished]
             Return (length)
```

Algorithm for STRING COPY operation

```
STR_CPY (str1, str2)
str1: string1
str2: string2
Step 1: [Initialization]
            i = 0
Step 2: [Perform copy]
            repeat while (str1[i] ≠ NULL)
            str2[i] = str1[i]
            i = i + 1
Step 3: [Terminate copied string]
           str2[i] = NULL
Step 4: [Finished]
```

Algorithm for STRING CONCAT operation

```
STR_CONCAT (str1, str2)
str1: Given string1
str2: Given string2
Step1: [Initialization]
        i = 0
        j = 0
Step2: [ Process until end of string1]
        repeat while (str1[i] ≠ NULL)
        str3[i] = str1[i]
        i = i + 1
Step 3: [ Process until end of string2]
        repeat while(str2[j] ≠ NULL)
        str3[i] = str2[j]
        i = i+ 1
        j = j + 1
Step 4: [Terminate string3]
        str3[i] = NULL
Step 5: [Finished]
return(str3)
```

Algorithm for STRING COMPARE operation

```
STR_CMP (str1, str2)
str1: Given string1
str2: Given string2
Step 1: [Initialization]
        i = 0
Step 2: [Find length of two strings]
length1 = strlen(str1)
length2 = strlen(str2)
Step 3: [Compare length of two strings]
if (length1 ≠ length2)
then
write ("Strings are not same")
return ()
Step 4: [Processed comparison]
repeat while (I < length1)
if (str1[i] ≠ str2[i])
then
write ("Strings are not same")
return ()
else
i= i + 1
Step 5: [return equal strings]
write ("strings are same")
return ()
```

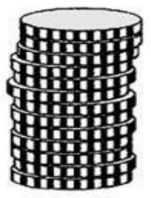
Stack and Queue

Q.1 Explain stack data structure in detail.

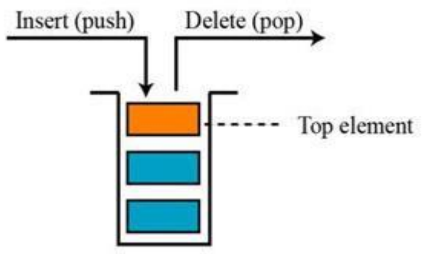
Stack

Stack is a linear list in which insertion and deletion operations are performed at only one end of the list. A stack is generally implemented with only two operations: PUSH and POP. The insertion operation is referred to as a PUSH operation. The deletion operation is referred to as a POP operation.

A pointer TOP keeps track of the top element in stack. Initially when the stack is empty, TOP has a value of zero and when the stack contains a single element: TOP has a value of one and so on. Each time a new element is inserted in the stack the TOP pointer is incremented by one and the pointer is decremented by one when element is deleted. Example – pile of tray in cafeteria.


Stack of coins

Stack of books



Computer stack

Q.2 Explain Push and Pop operation of stack with algorithm.

PUSH Operation

To insert an element on the TOP of the stack is called PUSH operation.

Algorithm for PUSH Operation

```
PUSH(S, TOP, X)
This algorithm inserts an element on top of the stack.
S represents stack.
TOP is a pointer which points to the top of the stack.
Step 1: [check for stack overflow]
        if (TOP>=N)
        then write ("Stack is Overflow")
        return ()
Step 2: [Increment TOP]
        TOP ← TOP +1
Step 3: [Insert element]
        S[TOP] ← X
Step 4: [finished]
        return ()
```

POP Operation

To remove an element from the TOP of the stack is called POP operation.

Algorithm for POP Operation

```
POP (S, TOP)
This algorithm removes an element from the top of the stack.
S represents stack.
TOP is a pointer which points to the top of the stack.
Step 1: [Check for stack underflow]
        If (TOP = 0)
        then write("Stack is underflow")
        return (0)
Step 2: [Delete an element]
        Y ← S[TOP]
Step 3: [Decrement pointer]
        TOP ← TOP - 1
Step 4: [return deleted element]
        return(Y)
```

Q.3 Explain applications of stack.

Applications of Stack

Three main applications of stack are: (1) Recursion (2) Stack machines (3) Polish notation. The recursive function is called function call to itself. Stack machine provides faster execution of polish notation. Better used for stacking local machines. It is used for representation of arithmetic expression. The process of writing the operators of an expression either before those operands or after operands are called the Polish Notation. There are basically three types of polish notation: Infix, Prefix, Postfix

Recursion

Recursion is a problem solving approach in which a problem is solved using repeatedly applying the same solution to smaller instances. Recursive function is a programming technique that allows the programmer to express operations in terms of themselves. The recursive function is called function call to itself.

Stack machines

Stack machine provides faster execution of polish notation. Better used for stacking local machines.

Polish notation

It is used for representation of arithmetic expression. The process of writing the operators of an expression either before those operands or after operands are called the Polish Notation. There are basically three types of polish notation: Infix, Prefix, Postfix

Q.4 Explain recursive functions with example.

Recursive Functions

Recursion is a problem-solving approach in which a problem is solved using repeatedly applying the same solution to smaller instances. Recursive function is a programming technique that allows the programmer to express operations in terms of themselves. The recursive function is called function call to itself.

Factorial of Number

The factorial function can be recursively defined as,

```
Factorial(n) = 1                        if n = 0
             = n * factorial(n-1)       if n > 0
```

Fibonacci series

The function of Fibonacci series can be recursively defined as,

```
Take n as input
if n=0 or n=1 then, fibo(n) = 0
else if n=2 then, fibo(n) = 1
else fibo(n) = fibo(n-1) + fibo(n-2)
```

Greatest Common Divisor

The function of GCD can be recursively defined as,

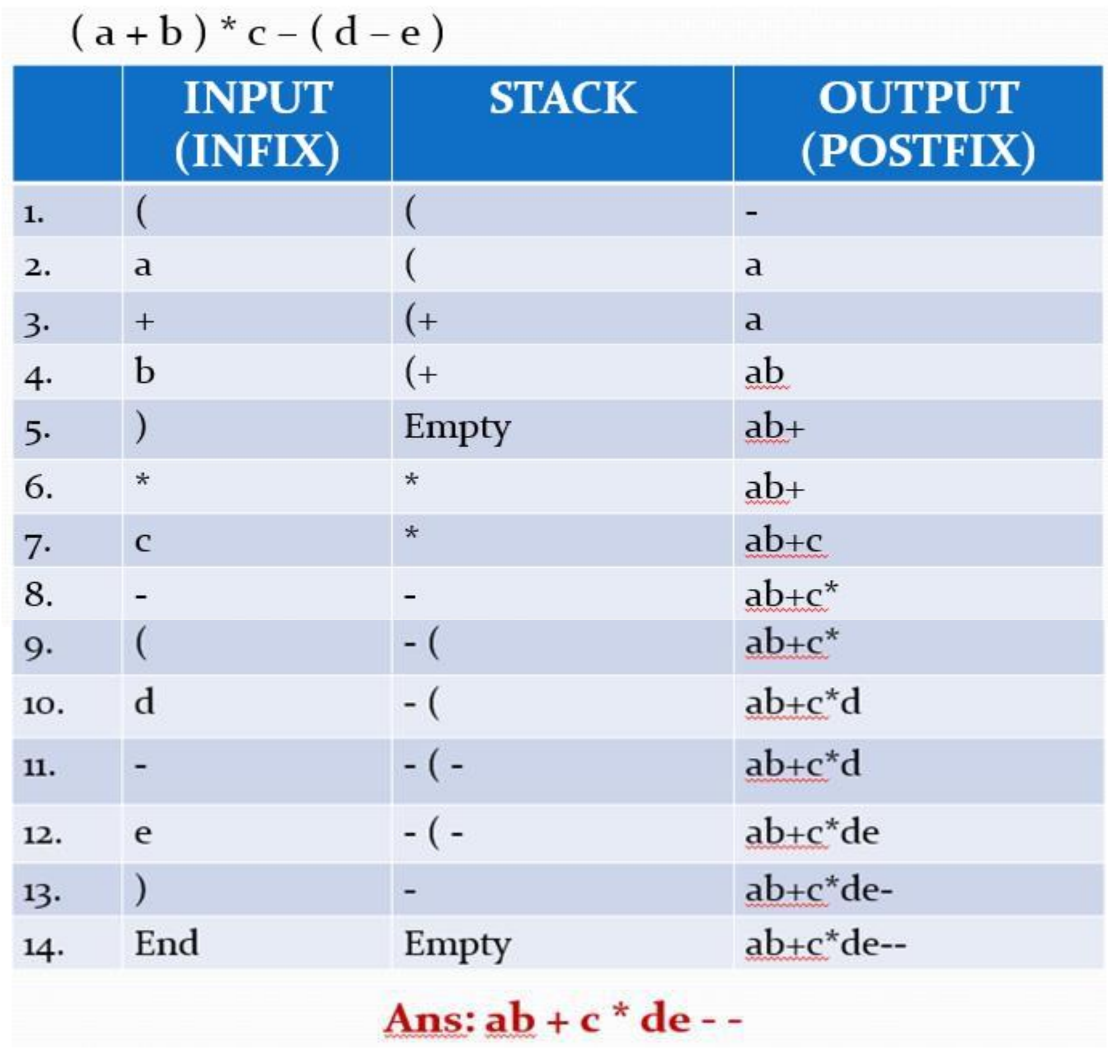
```
Take two inputs a and b
In case of a ≠ b
if a > b then, gcd(a,b) = gcd(a-b, b)
else if a < b then, gcd(a,b) = gcd(a, b-a)
In case of a = b, gcd(a,b) = a or b
```

Q.5 Explain steps to convert infix expression to postfix expression using stack.

1) Print operands as they arrive.
2) If stack is empty or contains a left parenthesis on TOP, push the incoming operator onto the stack.
3) If incoming symbol is '(', push it onto stack.
4) If incoming symbol is ')', pop the stack and print operators until left parenthesis is found.
5) If incoming symbol has higher precedence then the TOP of the stack.
6) If incoming symbol has lower precedence then the TOP of the stack, pop and print the TOP. Then test the incoming operator against the new TOP of the stack.
7) If incoming operator has equal precedence with the TOP of the stack, use associativity rule.
8) If expression ends then pop and print all operators of stack.

Q.6 Explain conversion from Infix to Postfix Expression for (a + b) * c – (d - e)

Conversion from Infix to Postfix Expression



(a + b) * c – (d – e)

	INPUT (INFIX)	STACK	OUTPUT (POSTFIX)
1.	(	(	-
2.	a	(	a
3.	+	(+	a
4.	b	(+	ab
5.	)	Empty	ab+
6.	*	*	ab+
7.	c	*	ab+c
8.	-	-	ab+c*
9.	(	- (	ab+c*
10.	d	- (	ab+c*d
11.	-	- (-	ab+c*d
12.	e	- (-	ab+c*de
13.	)	-	ab+c*de-
14.	End	Empty	ab+c*de--

Ans: ab + c * de - -

Q.7 Explain Queue data structure in detail.

Queue

A queue is a linear list of elements in which deletion can take place only at one end, called the FRONT, and insertions can take place only at the other end, called the REAR. Queue is also called First-in-First-out (FIFO) lists.

The term “front” and “rear” are used in describing a linear list only when it is implemented as a queue. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enters a queue is the order in which they leave.

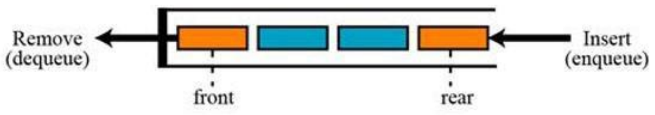
Real life examples of Queue are a row of students at a registration center, a movie ticket window example, line of cars waiting to proceed at traffic signal etc.

There are main two ways to implement a queue:
1. Circular queue using array
2. Linked Structures (Pointers)

Primary queue operations:
Enqueue: insert an element at the rear of the queue
Dequeue: remove an element from the front of the queue.

- Figure of Queue:

A queue of people



A computer queue

Q.8 Define Circular Queue data structure.

Circular Queue data structure
A Circular queue is a queue in which data are arranged such that the first element in the queue follows the last element in the queue. When we are deleting an element from simple queue, front pointer is incremented by one and the previous place of front pointer becomes useless. If rear pointer reaches to last element, we cannot insert any more elements. So, wasting of memory is there, this problem is solved by Circular queue.

Q.9 Explain operations of Simple Queue data structure.

Operations on Simple Queue

Insertion in Queue

To insert an element at the REAR end is called INSERTION operation.

Algorithm for INSERT operation

Q_INSERT (Q, F, R, N, Y)

```
This algorithm inserts an element into the queue.
Q represents queue vector containing N elements.
F and R are pointers pointing to the FRONT and REAR end.
Initially F and R are set to 0.
Step 1: [check for Queue overflow]
        if R ≥ N
        then write("Queue is Overflow")
        return (0)
Step 2: [Increment REAR pointer]
        R ← R +1
Step 3: [Insert element]
        Q[R] ← X
Step 4: [Set FRONT pointer]
        if F=0
        then F ←1
        return ()
```

Deletion in Queue

To remove an element at the FRONT end is called DELETION operation.

Algorithm for DELETE operation

```
Q_DELETE (Q, F, R, N)
Q represents queue vector containing N elements.
F and R are pointers pointing to the FRONT and REAR end.
Step 1: [Check for Queue underflow]
        if F = 0
        then write("Queue is underflow")
        return (0)
Step 2: [Delete element]
        Y ← Q[F]
Step 3: [Check for Queue empty]
        If F = R
        then F ← 0
             R ← 0
        Else
        F ← F + 1
Step 4: [return element]
        return(Y)
```

Q.10 Explain operations of Circular Queue data structure.

Operations on Circular Queue

Insertion in Circular Queue

To insert an element at the REAR end is called INSERTION operation.

Algorithm for INSERT operation

```
CQ_INSERT (Q, F, R, N, Y)
This operation inserts an element in the circular queue.
Q represents Queue vector containing N elements.
F and R are pointers pointing to the FRONT and REAR end.
Y is the element to be inserted.
Initially F and R are set to 0.
Step 1: [Reset rear pointer]
        if R >= N
        then R ← 1
        Else R ← R + 1
Step 2: [Overflow?]
        if F = R
        then write("QUEUE OVERFLOW")
        return ()
    Step 3: [Insert element]
        Q[R] ← Y
  Step 4: [Is front pointer properly set?]
        if F=0
        then F ← 1
        return ()
```

Deletion in Circular Queue
To remove an element at the FRONT end is called DELETION operation.

Algorithm for DELETE operation

CQ_ DELETE (Q, F, R, N, Y)

This operation deletes an element ad returns it from circular queue.

Q represents Queue vector containing N elements.

F and R are pointers pointing to the FRONT and REAR end.

Y is temporary variable.

Initially F and R are set to 0.

```
Step 1: [Check for Queue underflow]
        if F = 0
        then write("Queue is underflow")
        return (0)
Step 2: [Delete element]
        Y ← Q[F]
Step 3: [Check for Queue empty]
        if F = R
        then F ← 0
        R ← 0
        return(Y)
Step 4: [Increment front pointer]
        if F = N
       then F ← 1
        else F ← F + 1
        return(Y)
```

Q.11 Explain applications of Simple Queue data structure.

Applications of Queue

A queue is the natural data structure for a system to serve its incoming requests. Most of the process scheduling algorithm in operating system uses queues. Queues are used to schedule various jobs, tasks and processes for their execution by the CPU. Simulation is an application of Queue which allows experimenting without modifying the original situation.

Q.12 State the limitations of a Simple Queue and explain Circular Queue data structure.

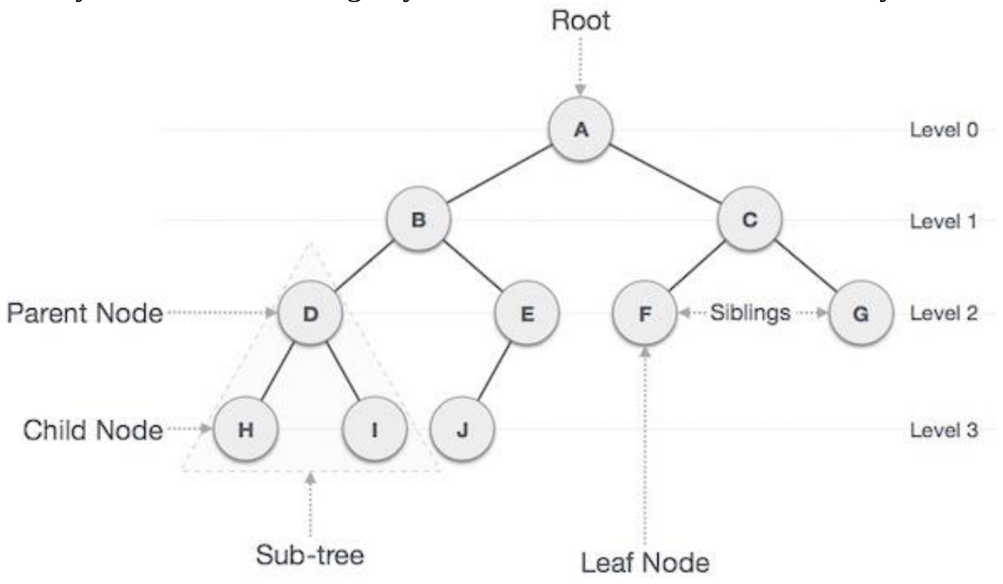
1) The limitation of simple Queue is that even if we have free spaces, we cannot use these memory spaces. So, simple queue results in wastage of memory. This problem can be solved by using Circular Queue.
2) A Circular queue is a queue in which data are arranged such that the first element in the queue follows the last element in the queue.
3) When we are deleting an element from simple queue, front pointer is incremented by one and the previous place of front pointer becomes useless.
4) If rear pointer reaches to last element, we cannot insert any more elements. So wasting of memory is there, this problem solved by Circular queue.
5) As by moving ahead, when front pointer or rear pointer reaches at last in the next move it moves to the first position.

Binary Tree

Tree

Definition

Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures. Tree is an example of non-linear data structure.
The Data Structure which is used to represent the hierarchical relationship between data element, with a special node named root and each node of data structure Stores a data value and has zero or more pointer to pointing to the other nodes which named ad child nodes, that data structure is called tree.
For example: Family trees, table of Contents, Directories Stored in Computer, Decimal Classification of books in library and records.
Trees are very useful in describing any Structure which involves hierarchy.



Terminology

Binary tree: The Binary tree is a special type of tree, which can be either empty or has finite set of nodes such that, one of the nodes is designated as root node and the remaining nodes are partitioned into two sub trees of root node known as left sub tree and right sub tree.
It is a tree in which each node can have maximum two children or you can say at most two children.it means each node can have 0,1 or 2 children but cannot have more than two children.

Indegree: Total number of entering vertices is known as indegree.

Out degree: In a directed graph, for any node v the number of edges which have v as their initial node is called the outdegree of node v.

Degree: In a tree the sum of edges Comes into particular node and edge Comes Out from Particular node is called degree of that Node.

Leaf Node: The nodes in the tree which have no child node or which are at the lowest level of the tree are called as leaf node.
Directed Edge: In a tree an edge which is given direction from onr Node to another Node then it is called Directed edges.

Complete binary tree: A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Root node: The node in the tree which is designated at the top of the tree and it has no parent node is called as Root Node.

Strictly binary tree: If every non-leaf node in a binary tree has nonempty left and right sub-trees, the tree is called a strictly binary tree.

Depth: The depth of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0. The height of a node is the number of edges on the longest path from the node to a leaf.

General Tree: General tree is a tree in which each node can have either zero or many child nodes.

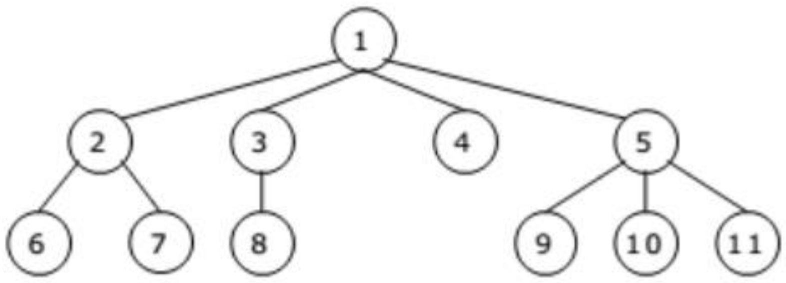
Edge: The line which Connect two Nodes is called as edges.

Siblings: The Nodes which belonging to the same parent Node are known as Siblings Node.

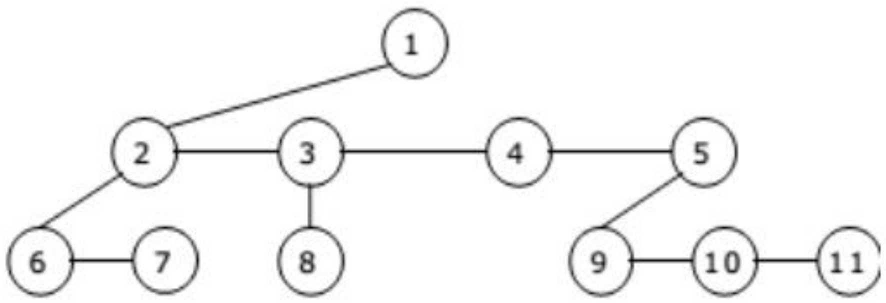
Conversion from General Tree to Binary Tree

Here some step for the Conversion:

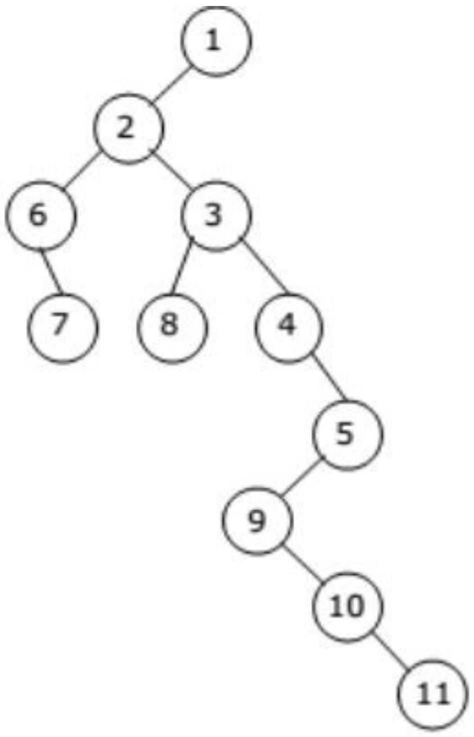
1. Find branch parent to left Most Child.
2. Connect Sibling of each Node L to R.
3. Delete All the Link.
4. Root of General Tree is Root of Binary tree.



Step 1:

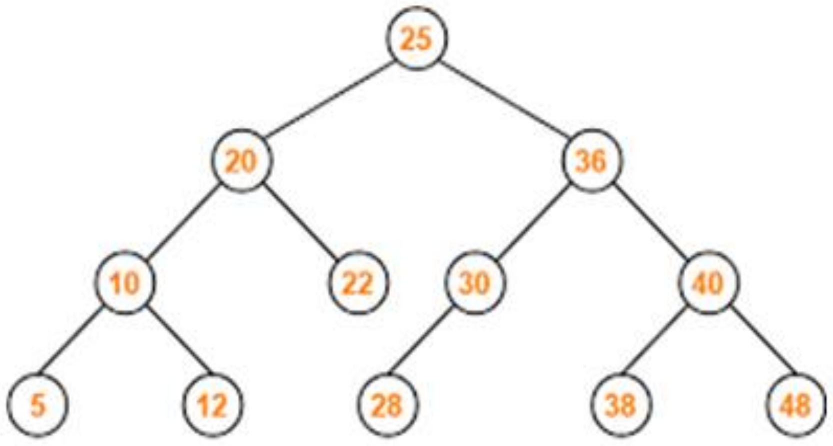


Step 2:



Binary search Tree

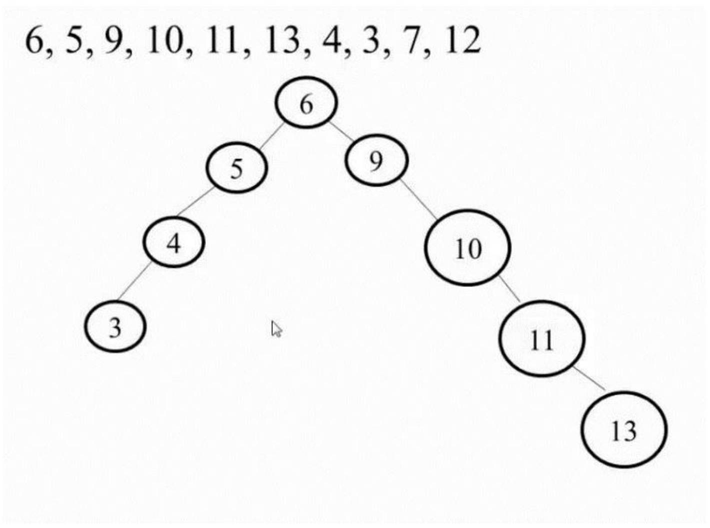
A binary search tree is a tree in which each node is arranged such that all the nodes in a left sub tree having values less then root and all the nodes in a right sub tree having values greater then root node.



Binary search tree having two characteristics:

1. All the nodes in a left sub tree having values less then root node.
2. All the nodes in a right subtree having values greater than root node.

Because of the above-mentioned characteristics binary search tree allow faster insertion deletion and searching facility



Operations on Binary search Tree

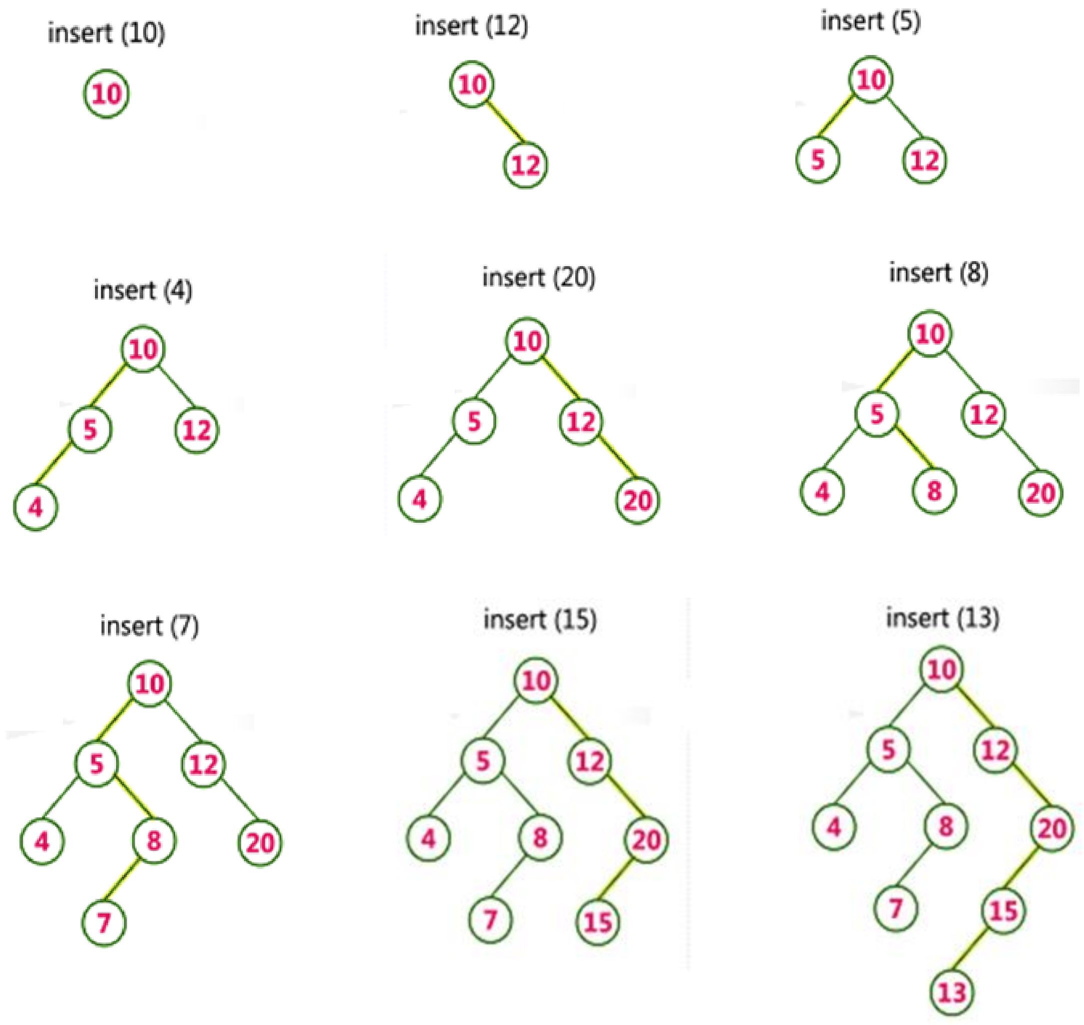
Insertion

Insertion of a Node in Binary search Tree:

1. All the nodes in a left sub tree having values less then root node.
2. All the Nodes in a right sub tree having values greater than root node.

Insertion of a Node in Binary search Tree:

1. All the nodes in a left sub tree having values less then root node.
2. All the Nodes in a right sub tree having values greater than root node.



Write an algorithm to insert new Node in Binary search tree

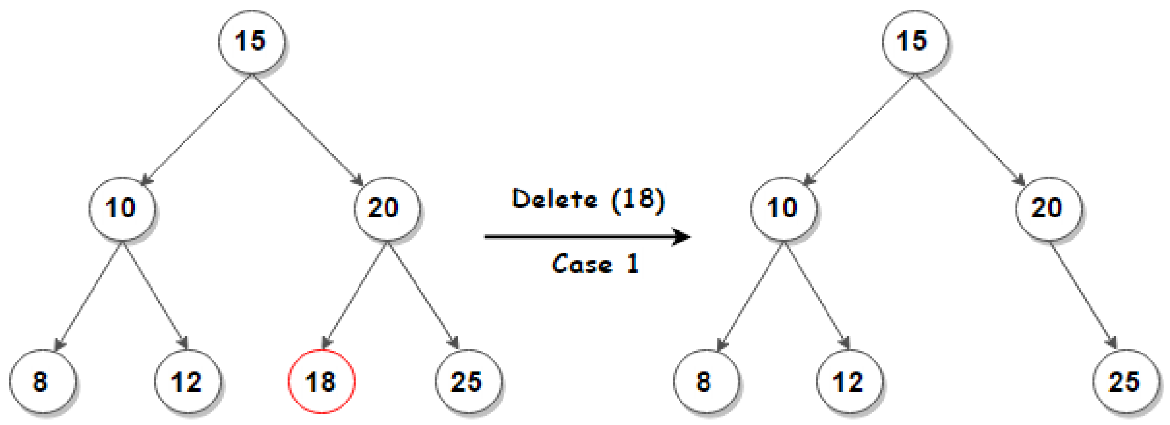
```
Step 1: IF AVAIL = NULL then
            Write “Availability Stack is Empty.”
        ELSE
            NEW_NODE = AVAIL
            AVAIL=AVAIL→RPTR
Step 2: If ROOT=NULL then
            NEW_NODE → INFO =X
            NEW_NODE → LPTR=NULL
            NEW_NODE → RPTR = NULL
            ROOT=NEW_NODE
            EXIT
Step 3: If X < ROOT→ INFO then
            Call INSERT (ROOT → LPTR, X)
        ELSE
            Call INSERT (ROOT → RPTR, X)
```

Deletion

In order to delete node from binary search tree we have to consider three possibilities.

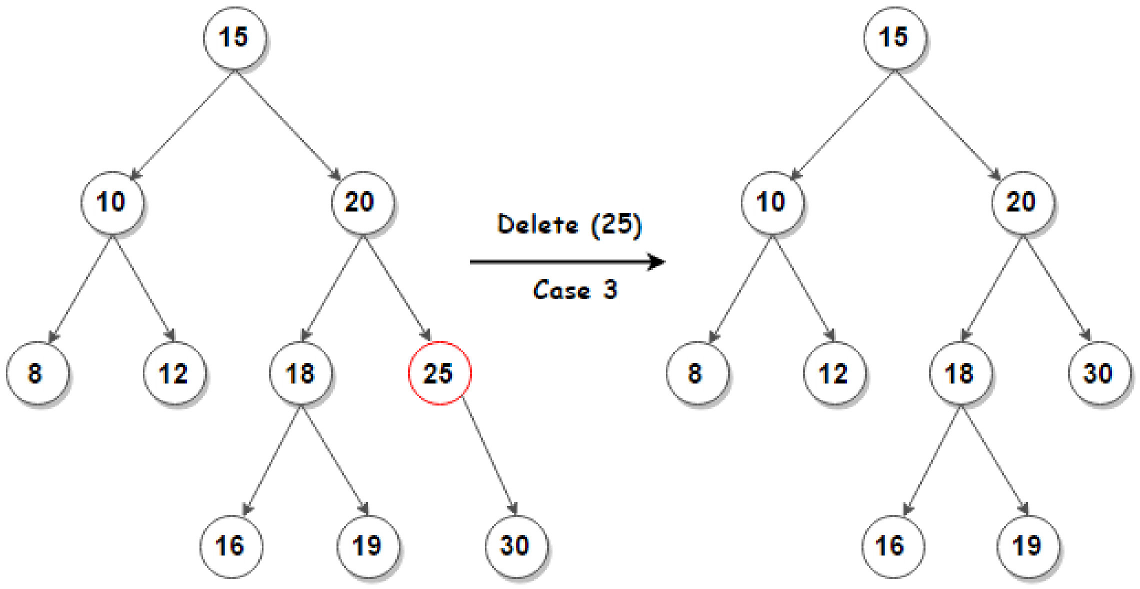
1. A node to be deleted has no sub tree
2. A node to be deleted has only one sub tree (left or right).
3. A node to be deleted has two sub trees (left and right)

Possibility 1:
It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space. In the following image, we are deleting the node 18, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



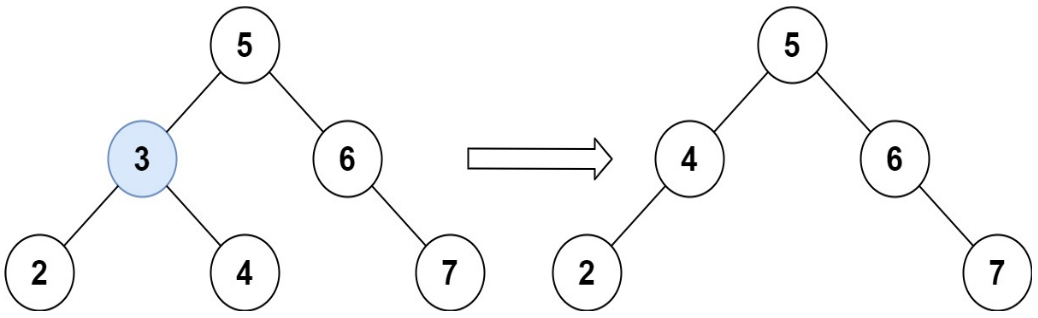
Possibility 2:
In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 25 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 25 (which is now leaf node) will simply be deleted.



Possibility 3:
If a node to be deleted has two sub trees left as well as right.in such case we have to perform following steps

1. Find in order child of the node to be deleted.
2. Append the right sub-tree of the in-order child to its grant parent.
3. Replace the node to be deleted with its in-order child.



Searching

In order to searching a node in binary search tree we have to follows the step given below.

Step-1
First, we have to check weather binary search tree is empty or not. If binary search tree is empty then search is un successful.

Step-2
If binary search tree is not empty then we compare the value of a node to be searched with root node of binary tree if both values are equal then search is Successful otherwise, we have two possibilities.

1. If the value of the node to be searched having value less than the value of root node then we have to search the node in left subtree of root node.
2. If the value of the node to be searched having value greater than the value of root node

then we search the node in sub tree of root node

Step 2 is repeated recursively until node to be searched is found or all the nodes in binary search tree are compared with the value of the node to be searched.

Algorithm for search operation on binary search tree

```
Step 1: If ROOT = NULL then
            Write "Tree is Empty. Search un-successful"
Step 2: If X=ROOT → INFO then
            Write "Search is successful"
            Return (ROOT)
Step 3: If X < ROOT → INFO then
            Call SEARCH (ROOT → LPTR, X)
            Else
            Call SEARCH (ROOT → RPTR, X)
```

Tree Traversal

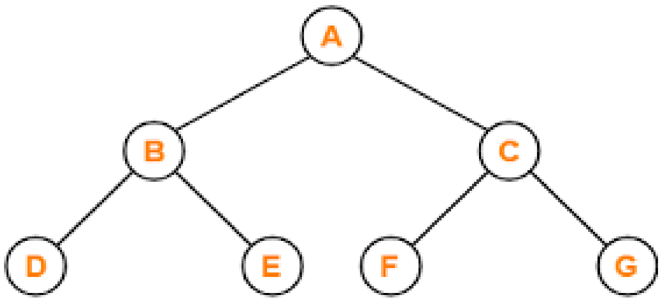
Traversal is the method of processing each and every node in the Binary Search Tree exactly once in Systemic manner. There are three different type of tree traversal.

1. Preorder Traversal
2. Inorder Traversal
3. Postorder Traversal

Pre-order Traversal

In Order to traverser the free in preorder follow the steps given below

1. Process the root node first.
2. Traverse the left sub tree in preorder.
3. Traverse the right sub tree in preorder.



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

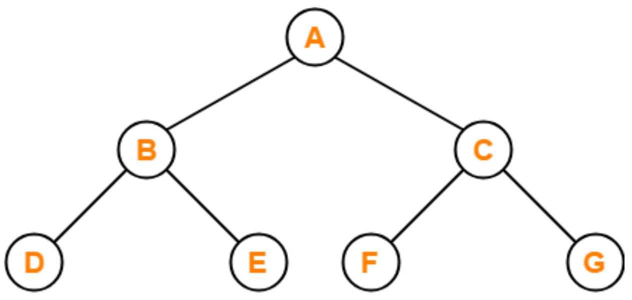
Algorithm for pre-order traversal

```
Step 1: If ROOT = NULL then
            Write "Tree is Empty"
            Else
            Write ROOT → INFO
Step 2: If ROOT → LPTR != NULL then
            Call PREORDER (ROOT → LPTR)
Step 3: If ROOT → RPTR != NULL then
            Call PREORDER (ROOT → RPTR)
```

In-order Traversal

In order to traverse the tree in in-order follow the steps given below:

1. Traverse the left sub-tree in in-order.
2. Process the root node.
3. Traverse the right sub-tree in-order.



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

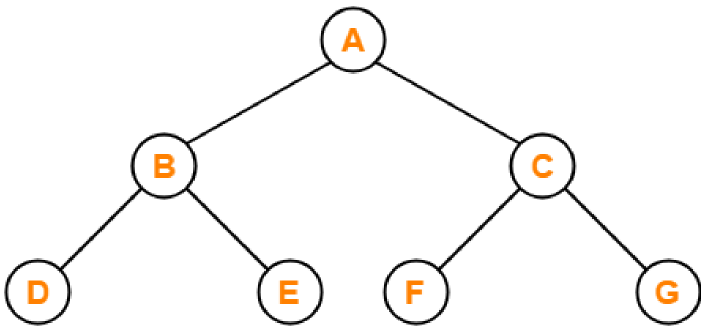
Algorithm for in-order traversal

```
Step 1: If ROOT = NULL then
            Write "Tree is Empty"
Step 2: If ROOT → LPTR != NULL then
            Call INORDER (ROOT → LPTR)
Step 3: Write ROOT → INFO
Step 4: If ROOT → RPTR != NULL then
            Call INORDER (ROOT → RPTR)
```

Post-order Traversal

In order to traverse the tree in post-order follow the steps given below.

1. Traverse the left sub tree in post-order.
2. Traverse the right sub tree in post-order.
3. Process the root node.



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

Algorithm for post-order traversal

```
Step 1: If ROOT = NULL then
            Write “Tree is Empty”
Step 2: If ROOT → LPTR != NULL then
            Call POSTORDER (ROOT → LPTR)
Step 3: If ROOT → RPTR != NULL then
            Call POSTORDER (ROOT → RPTR)
Step 4: Write ROOT → INFO
```

Application of binary tree

1. It is used in the Manipulation of arithmetic expressions.
2. It can be also used in the Construction and maintenance of symbol table.
3. Tree is widely used in compiler for syntax analysis.
4. Tree is widely used to represent more complex data Storage.

Sorting and Searching techniques

Q.1 Explain Sorting. List out different sorting methods.

Sorting:

Definition

Sorting is the process of arranging elements of a List in particular order either ascending or descending. The Sorting is performed according to the key value of each record. There are number of methods available for performing Sorting. Each one is differed from other. Also, the running time of each method is different, (it is the complexity of an algorithm).

In general, the main task in sorting is basically comparison and interchanging position of elements. There are various types of sorting method available. Sorting method are basically divided into two subcategories.

1. Internal Sort

The Sorting method that does not required external memory for sorting the elements is known as internal sort. It is useful when we must sort fewer amounts of elements.

Example:

1. Bubble Sort
2. Quick Sort
3. Selection Sort
4. Insertion Sort

2. External Sort

The sorting method that required external memory for sorting the element is known as external Sort. It is useful when we must sort large number of elements.

Example:

1. Merge Sort
2. Radix Sort

Q.2 Explain Bubble Sort Method.

Bubble sort is a simple sorting algorithm. Bubble Sort is an example of Internal Sort. Bubble Sort compares all the element one by one and sort them based on their values. If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times. It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Example:

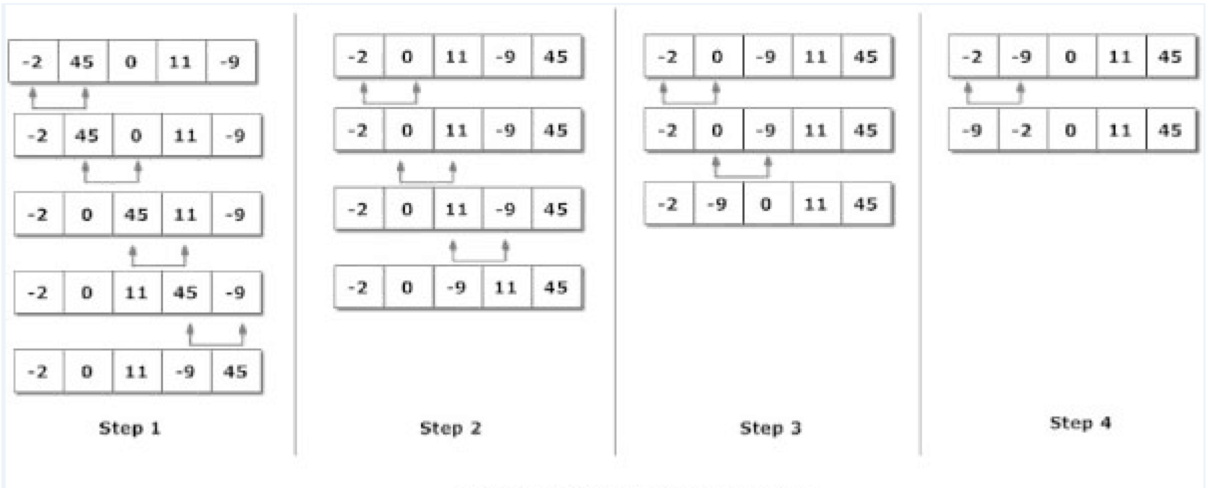


Figure: Working of Bubble sort algorithm

Algorithm:

Step 1: Repeat up to Step 3 for I=0 to N-1

Step 2: Repeat step 3 for J=0 to N-I-1

Step 3: If a[J] > a[J+1] then

    TEMP ➝ a[J]

    a [J] ➝ a[J+1]

    a[J+1] ➝ TEMP

Step 4: Exit

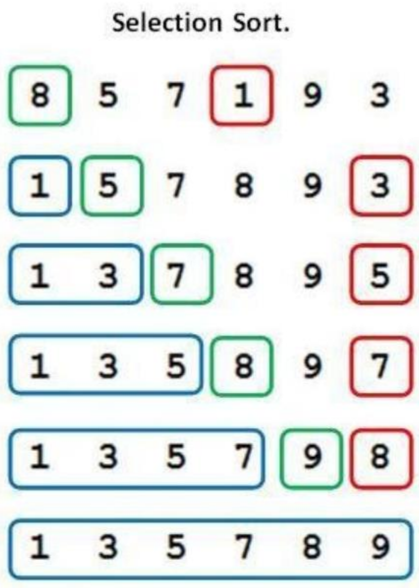
Q.3 Explain Selection Sort Method.

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty, and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst-case complexities are of $O(n^2)$, where **n** is the number of items.

Example:



Algorithm:

```
Step 1: Repeat up to Step 5 for i=0 to N-1
Step 2: Min ← a[i]
Pos ← i
Step 3: Repeat Step 4 for j=i+1 to N
Step 4: if a[j] < Min then
Min ← a[j]
Pos ← j
Step 5: TEMP ← a[i]
a[i] ← a[Pos]
a[Pos] ← TEMP
Step 6: Exit
```

Q.4 Explain Quick Sort.

Quick Sort method is an efficient Sorting method for large list. It works fine for the list having large number of elements. It uses divide and Conquers Strategy in which a list is divided into two smaller Lists.

First initialize Pivot with index of the list of elements, then Low with index of Pivot+1 of elements and high with index of last element.

Now we scan element from Left to right and compare each element with pivot element. If the scanned element is less, then the pivot element. We scan next element and increment the value of low. Repeat same Procedure until we found the element which is greater than the pivot element.

Now we scan elements from right to left and compare each element with Pivot element if the Scanned element we scanned elements is greater then the pivot elements we scan next element which is less than the Pivot element.

Now we compare the value of low and high. If Low is Less, then High then we interchange the element which are at the index Low and High.

Increment the value of low and decrement the value of high, Repeat above procedure while value of low less than or equal to value of high.

After the Completion of first PASS the entire list of elements is divided into two list first list Contains elements which are less then the pivot elements which are greater than the pivot element. The above Procedure is recursively repeated for the sub lists until all the elements in the lists are Sorted.

The order of Comparison for this method is $O(n\log_n)$.

Partitioning:

1) Pivot=First index
2) I=Pivot +1
3) J=Last index
4) I++ until element > Pivot is found
5) J - - until element <= Pivot is found

If (element is not found less, then Pivot) then

Pivot is on Correct Position then no need to Swap and do Pivot+1

6) Swap a[i] & a[j] and repeat step 4 & 5 until j<=i.
7) If(j<i) Swap pivot & a[j].

Algorithm: Quicksort (a [], low, high)

```
Step 1: if (low<high)
        partitionIndex ← partition(a[],low, high)
Step 2: Quicksort(a[],low,partitionIndex-1)
Step 3: Quicksort(a[],partitionIndex+1,high)
Step 4:Exit
Partition(a,low, high)
Step 1:
        pivot ← low
        i ← pivot+1
        j ← high
step 2: repeat step 3,4 and 5 until j<=i
step 3: i ← i+1 until
        a[i]>a[pivot]
step 4: j ← j+1until
        a[j] <=a[pivot]
step 5: swap a[i] & a[j]
step 6: if(j<=i)
        swap a[pivot] & a[j]
```

Q.5 Explain Insertion Sort.

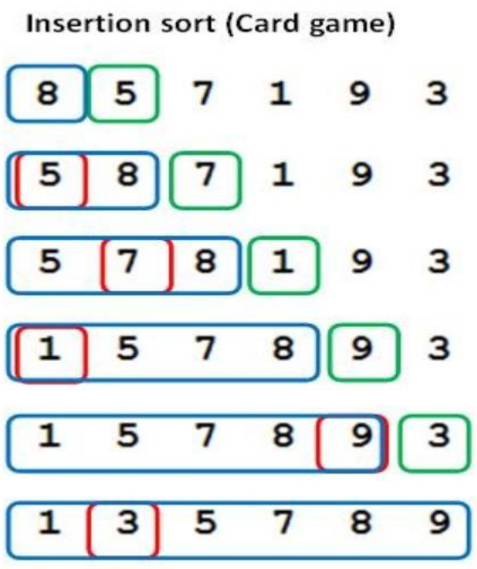
Insertion Sort method is a Simple Sorting method, a comparison sort in which the sorted list is built one entry at time.It is much less efficient on large list than more advanced algorithms like quick sort, heap sort or more sort.

However, insertion sort provides several advantages like:

- Simple to implement.
- Efficient for small data set.
- It is adaptive as more efficient for small data set
- It is adaptive, as more efficient when given data are sorted.
- It is stable, as it does not change the the relative order of elements with equal keys.
- It is online, because it can sort a list as it receives list.

A very simple example of insertion sort is when we are planning a game of cards, we are taking cards one by one and arranging them accordingly.

Example:



Algorithm:

```
Step 1: [initialize i]
        I ← 1
Step 2:[traverse through the list and compare elements]
        Repeat while (i<n)
                KEY ←  a[i]
                J←  i-1
                Repeat while (j <= 0 and KEY < a[j])
                        a[j+1] ←  a[j]
                j ←  j-1
                a[j+1] ← KEY
                I ←  i+1
Step 3: [finish]
        Return ()
```

Q.6 Explain Merge Sort.

The merge sort algorithm closely follows the divide-and-conquer paradigm. it operates as follows.

Divide: Divide the n-elements sequence to be sorted into two sub-sequences of n/2 elements each recursively.

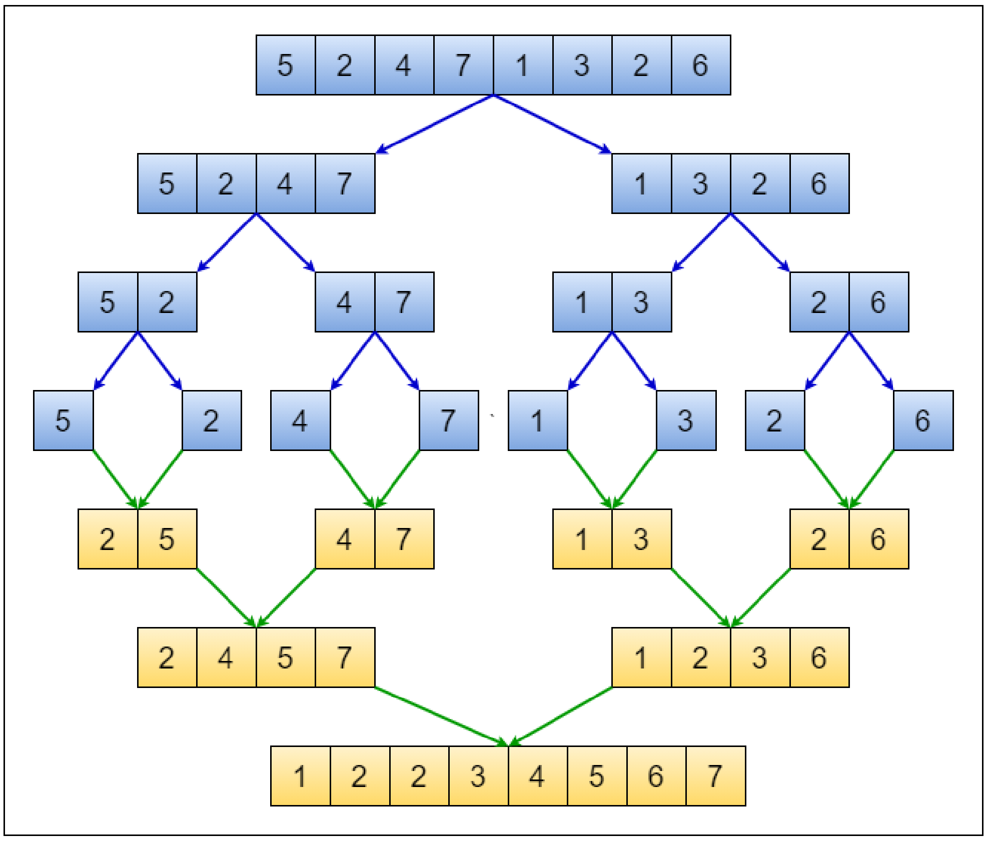
Conquer: Sort the two sub-sequences recursively using merge sort.

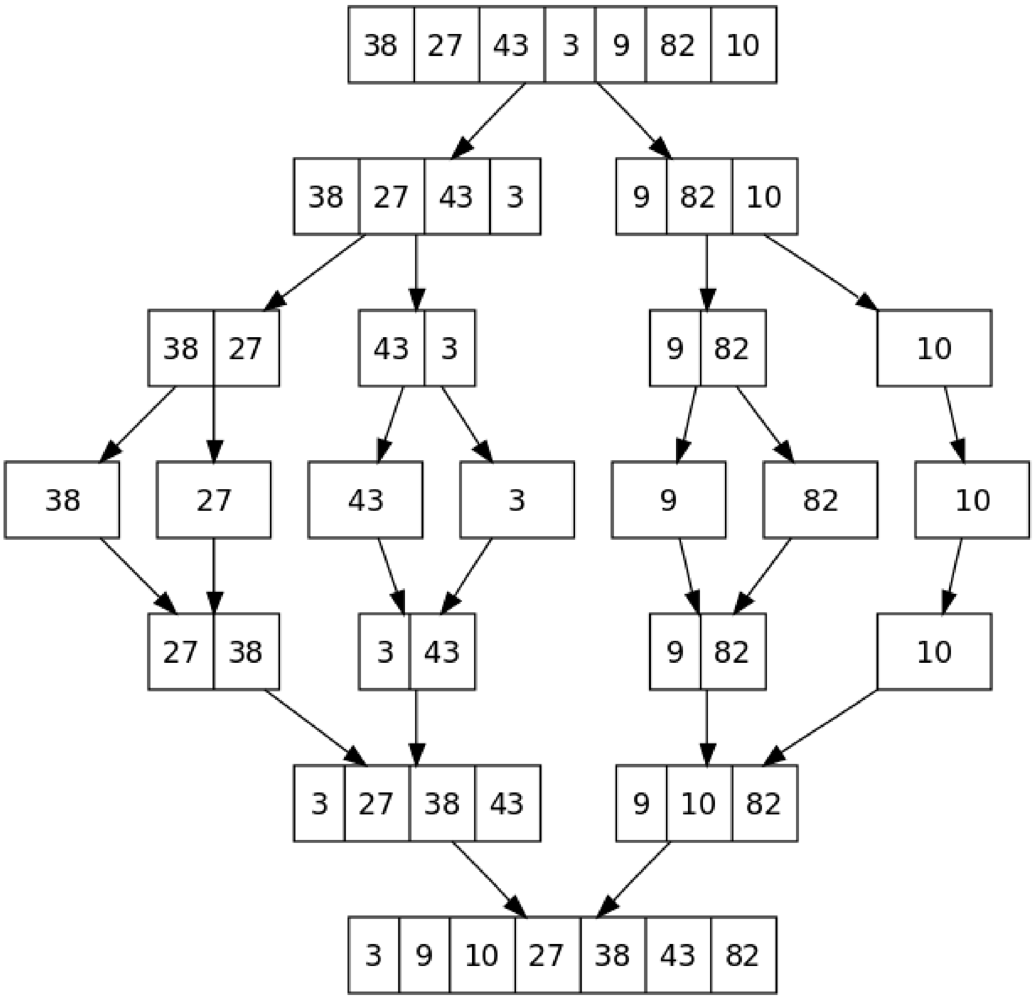
Combine: Merge the two sorted sub-sequences to produce the sorted answer.

Algorithm:

```
Mergesort(a,i,j)
Step 1:
        i ← low
        j ← high
Step 2:
        Mid ← (i+j)/2
        Mergesort(a,i,mid)
        Mergesort(a,mid+1,j)
Step 3:
        Merge(a, i, mid, j)
Algorithm: Merge(a, i, mid,j)
Step 1:
        i ← low
        j ← mid+1
        k ← low
Step 2:
        Repeat Step 3 for i<=mid and j<=high
Step 3:
        If(a[i] <=  a[j])
        b[k] ← a[i]
        i ← i++
else
b[k] ← a[j]
j ← j+1
Step 4:  k ← k+1
Step 5:
        if(i>mid)
                while(j<=high)
                b[k] ← a[j]
                j ← j+1
k ← k+1
Else
 while(i<=mid)
b[k] ← a[i]
i ← i+1
k ← k+1
step 6:  Exit
```

Example:
5,2,4,7,1,3,2,6





Q.7 Explain Radix Sort.

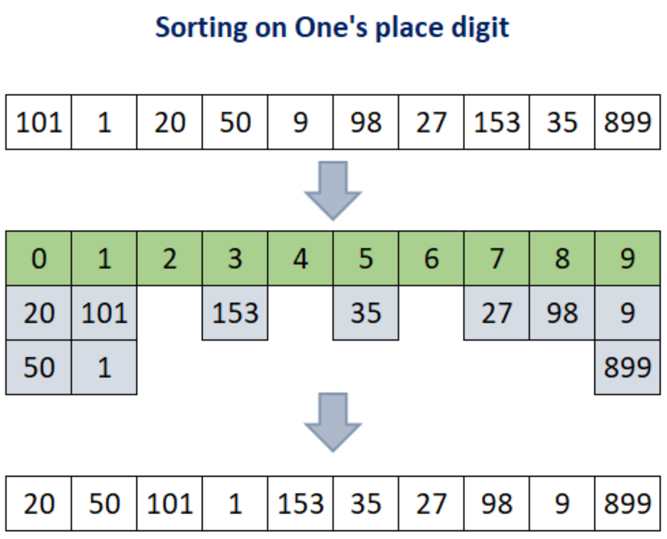
Radix sort is based on the idea that the sorting of the input data is done digit by digit from least significant digit to most significant digit and it uses counting sort as a subroutine to perform sorting.

Counting sort is a linear sorting algorithm with overall time complexity *Θ(N+K)* in all cases, where *N* is the number of elements in the unsorted array and *K* is the range of input data.

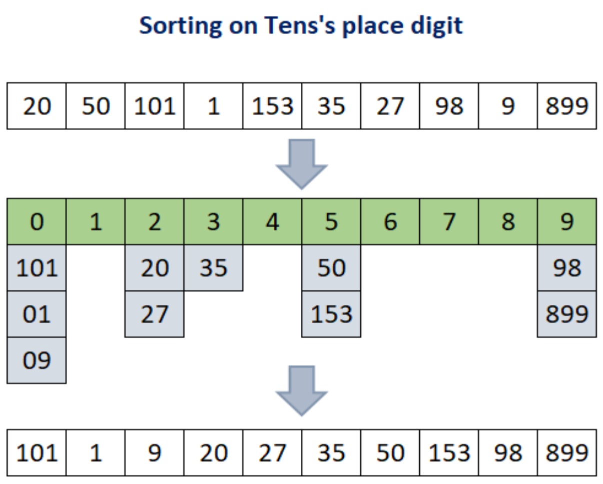
The idea of radix sort is to extend the counting sort algorithm to get a better time complexity when *K* goes up.

Example: lets consider an unsorted array *A = [101, 1, 20, 50, 9, 98, 27, 153, 35, 899]* and discuss each step taken to sort the array in ascending order.

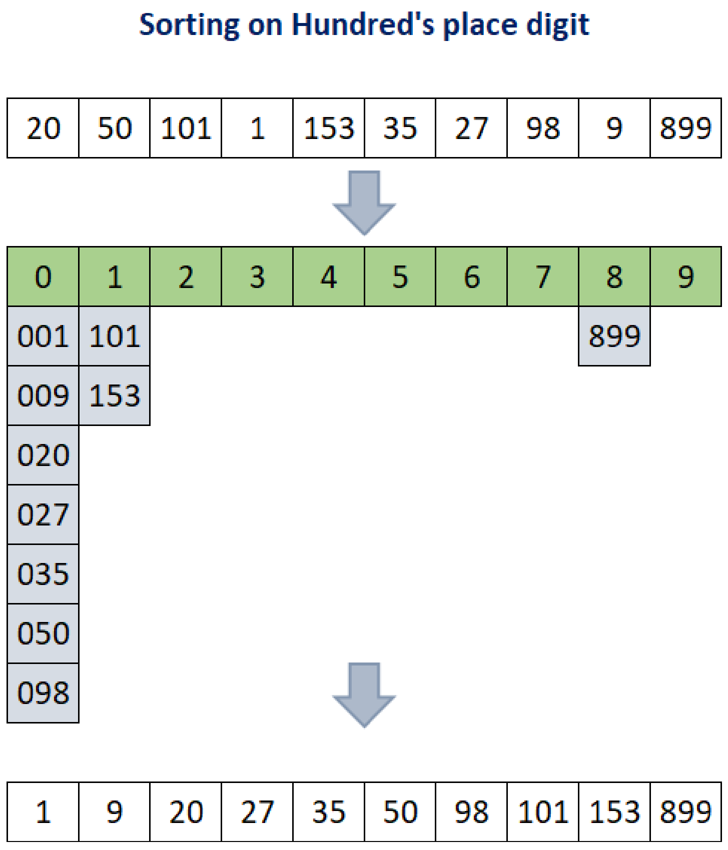
Step 1:



Step 2:-



Step 3:-



Q.8 Explain following terms with example.

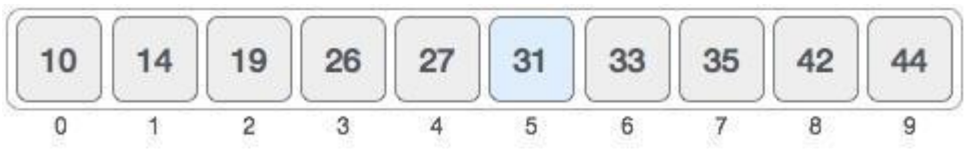
(a) Binary search method

(b) Sequential search method

Binary search method:

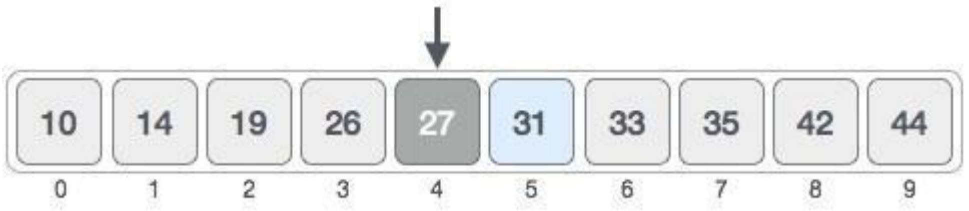
Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues the sub-array as well until the size of the sub-array reduces to zero.

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



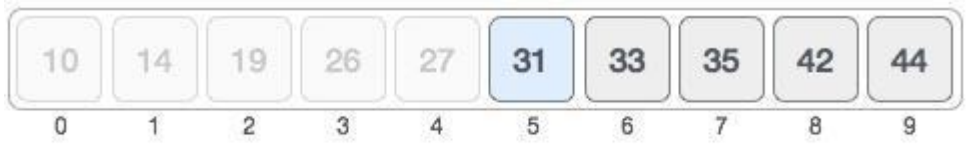
First, we shall determine half of the array by using this formula

Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e., 31. We find that the value at location 4 is 27, which is not a match.

As the value to be searched is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

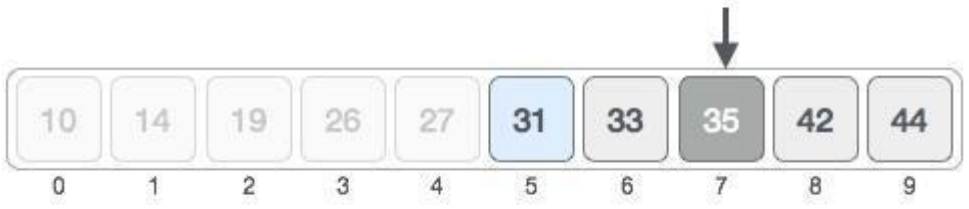


We change our low to mid + 1 and find the new mid value again. low = mid + 1

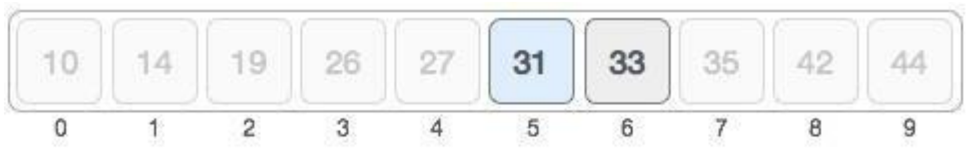
mid = (low + high) / 2

Here it is, (5 + 9) / 2 = 7

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

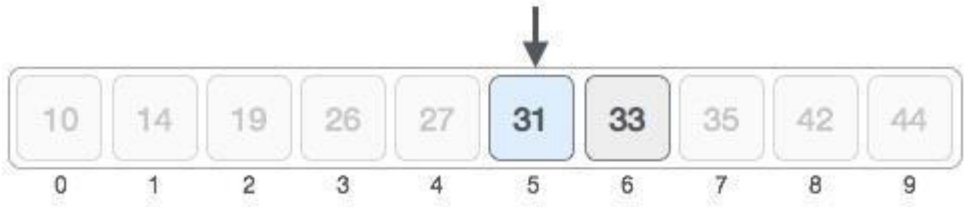


Hence, We change our low to mid -1 and find the new mid value again.

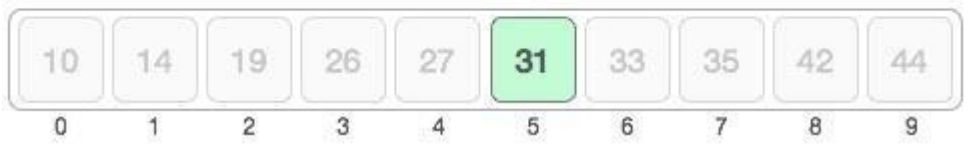
low = mid + 1

mid = (low + high) / 2

Here it is, (5 + 6) / 2 = 5 (integer value of 5.5). So, 5 is the mid of the array.



We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search find the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Sequential search method

Sequential search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise, the search continues till the end of the data collection. The following is our sorted array and let us assume that we need to search the location of value 33 using sequential search.

We conclude that the target value 33 is stored at location 6.

Q.9 Write an algorithm for following searching method

Algorithm : BINARY_SEARCH(List, n, x)

List : Array

n : Size of Array

x : Element to be searched

```
Step 1:[Initialization]
        low  ← 0
        high  ← n-1
        flag   ←1
Step 2:
        while(low <= high)
        repeat step 3 to 4
Step 3:
        mid = (low+high)/2
Step 4:
        if (x < List[mid])
        then high  = mid-1
        else if (x > List[mid])
        then low= mid+1
        else if(x  = List[mid])
        then
        write("Search  is successful  at location  : mid+1)
        flag  ← 0
        exit
Step 5:
        if ( flag  = 1)
        then Write("Search  is unsuccessful")
Step 6:[Finished]
        exit
```

Algorithm : SEQUENTIAL_SEARCH(List, n, x)

List : Array

n   : Size of Array

x   : Element to be searched

Step 1: [Initialization]

```
        i ← 0
        flag  ←1
Step 2:
        repeat step 3 for i = 0, 1, 2, … , n-1
Step 3:
        if(x  = List[i])
        then write(“Search  is successful  at location  : i + 1”)
        flag  ← 0
exit
Step 4:
        if (flag  =1)
        then Write(“Search  is unsuccessful”)
Step 5: [Finished]
exit
```