DATA STRUCTURE

Third Semester
Theory-2

Prepared By
D Susmita
Sr. Lect.(IT)

Ch 1 Introduction

What is data?

Data is a representation of Facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing by humans or by automatic means.

Is number a data?

Is word a data?

What is information?

Processed data is called information.

What is a data type?

Data types are used within type systems, which offer various ways of defining, implementing and using the data.

Almost all programming languages explicitly include the notion of data type. Different languages may use different data type terminology. Common data types may include:

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

Classes of data types

There are different classes of data types as given below.

- **Primitive data type:** primitive Data Types are **built-in data types** defined by any language specification.

- **Composite data type:** A composite data type is **one which is composed with various primitive data type**

- **Boolean data type:**   The BOOLEAN data type stores TRUE or FALSE data values as a single byte..... You can compare two BOOLEAN values to test for equality or inequality. You can also compare a BOOLEAN value to the Boolean literals ' t ' and ' f '. BOOLEAN values are not case-sensitive; ' t ' is equivalent to ' T ' and ' f ' to ' F '.

- **En-numerated data type :** : Enumeration is **a user defined datatype** in C language. It is used to assign names to the integral constants which makes a program easy to read and maintain.

```c
#include<stdio.h>
enum week{Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun};
enum day{Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund};
int main() {
  printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",Mon , Tue, Wed, Thur, Fri, Sat, Sun);
  printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",Mond , Tues, Wedn, Thurs, Frid, Satu, Sund);
  return 0;
}
```

Output

The value of enum week: 10111213101617

The default value of enum day: 0123181112

- **Abstract data type:** An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

An ADT is a mathematical model of a data structure that specifies the type of data stored, the operation performed on data & parameters of the operation.

ADT specifies what each operation does but not how it does it.

For example, a List is an abstract data type that is implemented using a dynamic array and linked list. A queue is implemented using linked list-based queue, array-based queue, and stack-based queue. A Map is implemented using Tree map, hash map, or hash table.

Let's understand the abstract data type with a real-world example.

If we consider the Smartphone. We look at the high specifications of the Smartphone, suchas:

- o 4 GB RAM
- o Snapdragon 2.2ghz processor
- o 5 inch LCD screen
- o Dual camera
- o Android 8.0

The above specifications of the Smartphone are the data, and we can also perform the following operations on the Smartphone:

- o **call():** We can call through the Smartphone.
- o **text():** We can text a message.
- o **photo():** We can click a photo.
- o **video():** We can also make a video.

The Smartphone is an entity whose data or specifications and operations are given above. The abstract/logical view and operations are the abstract or logical views of a Smartphone.

What is Data Structure?

The logical & mathematical model of a particular organization of data is called data structure.

A data structure is a technique of organizing the data so that the data can be utilized efficiently. There are two ways of viewing the data structure:

- o **Mathematical/ Logical/ Abstract models/ Views:** The data structure is the way of organizing the data that requires some protocols or rules. These rules need to be modeled that come under the logical/abstract model.
- o **Implementation:** The second part is the implementation part. The rules must be implemented using some programming language.

The choice of data model depends on two considerations:

1. It must be rich enough to mirror the actual relationships of the data in the real world.

2. The structure should be simple that can be effectively processed when necessary

A data structure is a specialized format for **Organizing, Processing, Retrieving and Storing** data.

Operations of data structure:

There are six operations performed on data in data structure such as

1. **Traversing:-**Visiting & Processing only once.

2. **Insertion:-**Addition of a new element in the list.

3. **Deletion:-**Deletion of an existing element from the list.

4. **Searching:-**Finding the location of existing element in the list.

5. **Sorting:-**Arrangement of existing elements in the list either in ascending /descending order.

6. **Merging:-**Combing two lists & make it as one list.

Algorithm:

An algorithm is a set of instructions for solving logical and mathematical problems, or for accomplishing some other task.

- **Unambiguous – Algorithm should be clear and unambiguous. ...**

- **Input – An algorithm should have 0 or more well-defined inputs.**

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

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

- **Feasibility** – Should be feasible with the available resources.

- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code

Complexity of Algorithms:

Suppose **M** is an algorithm & **n** is the number of input data.
The **Time** & **Space** used by the algorithm **M** are the two measures for efficiency of **M.**

The time is measured by counting the number of key operations(Number of comparisons).

The space is measured by counting the maximum amount of memory needed by the algorithm.

Complexity of an algorithm **M** is the function **f(n)** which gives the running time/storage space requirement of the algorithm in terms of the size **n** of the input data.

Frequently, the storage space required by an algorithm is simply a multiple of if the data size '**n'.**

Unless otherwise stated or implied, the term "Complexity" shall refer to the running time of the algorithm.

The efficiency of M is measured in terms of time and space used by the algorithm. Time is measured by counting the number of operations and space is measured by counting the maximum amount of memory consumed by M. The complexity of M is **the function f(n) which gives running time** and or space in terms of **n**.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is $T(n) = c * n$, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Time-space tradeoff :

In computer science, a space-time or time-memory trade-off is **a way of solving a problem or calculation in less time by using more storage space (or memory)**, or by solving a problem in very little space by spending a long time.

It is a situation where one thing increases & other thing decreases. It is a way to solve a problem either in less time by using more space or in less space by spending a long amount of time.

Finally, time complexity is the time taken by executing the algorithm of each instruction.

Space complexity is referred to as the amount of memory consumed during execution of algorithm.

Assignment:1

1. what is data? What is information?
2. Define Data structure.
3. Explain any four datatypes in data structure.
4. What is algorithm. Write its characteristics.
5. Explain Space Complexity.
6. Explain time Complexity.
7. Explan time-space trade-off.

Programming in c
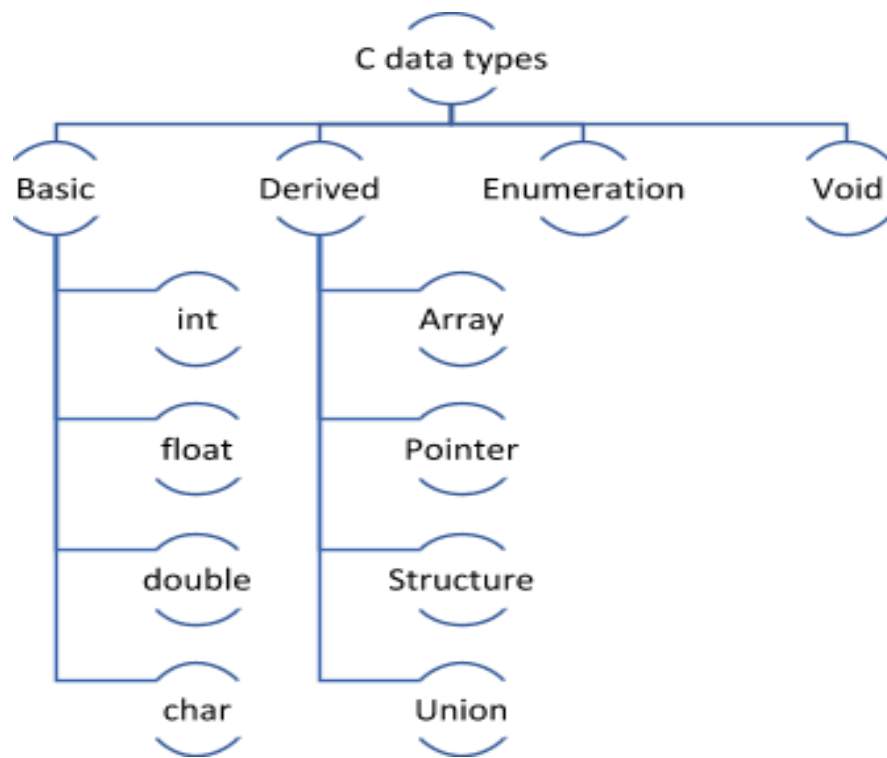
1. Ex in c: int ,char,float,double

```c
double average = 679999999.454;

float score = 679999999.454;

printf("average is %lf", average);

printf(", score is %f", score);

char group = 'B';

To print a name or a full string, we need to define char array.

 char group = 'B';

char name[30] = "Student1";

printf("group is %c, name is %s", group, name);
```

```c
signed char char1 = -127;

unsigned char char2 = -127;

printf("char1 is %d, char2 is %d", char1, char2);
```

Derived Data Types

Array, pointers, struct, and union are the derived data types in C.

```c
#include

int main(void) {

 // declare array with maximum 5 values

 int marks[5];

 // get the size of the array

 int noOfSubjects = sizeof(marks)/sizeof(int);
```

```c
// let us get the inputs from user

for(int i=0; i<noOfSubjects; i++)

{

printf("\nEnter marks ");

scanf("%d", &marks[i]);

}

double average;

double sum = 0;

// fetch individual array elements

for(int i=0; i<noOfSubjects; i++)


// let us print the average of marks

average = sum/noOfSubjects;

printf("\nAverage marks = %lf", average);

return 0;

}
```

Pointer:

```c
#include

int main(void) {

int  *ptr1;

int  *ptr2;

int a = 5;

int b = 10;

/* address of a is assigned to ptr1*/
```

```c
    ptr1 = &a;

    /* address of b is assigned to ptr2*/

    ptr2 = &b;

    /* display value of a and b using pointer variables */

    printf("%d", *ptr1); //prints 5

    printf("\n%d", *ptr2); //prints 10

    //print address of a and b

    printf("\n%d", ptr1); // prints address like -599163656

    printf("\n%d", ptr2); // prints address like -599163652

    // pointer subtraction

    int minus = ptr2 - ptr1;

    printf("\n%d", minus); // prints the difference (in this case 1)

    return 0;

}
```

Structs

```c
int main(void) {

    // Store values in structures

    Student st1 = {"student1", 1, 'a', , 4.5};

    Student st2 = {"student2", 2, 'b', , 9.5};

    // Send structure values to the printing method

    print_student_details(&st1);

    print_student_details(&st2);

    return 0;

}

// get the address of structure data and print
```

```c
void print_student_details(Student *st) {

  printf("\Student details for %s are:\n", st->name);

  printf("id: %d\n",st->id);

  printf("group %c\n", st->group);

  // since marks is an array, loop through to get the data

  for(int i=0;i<5;i++)

  printf("marks %f\n", st->marks[i]);

  printf("interest %lf", st->interest);

}
```

Programming assignment in C

1. wap in c to display the alternate elements from a two dimentionalarrayi.e. 3*3

2. wap in c to reverse the entire content of the arr having N elementswithout using any other array.

3. wap in c to display the content of a two dimentional array witheach column content in reverse order.

4. wap in c to print the product of each row of a 2D integer array.

5. wap in c to to find the sum of diagonal elements from a 2D arrayof integer.

2.0 STRING PROCESSING   03
2.1 Explain Basic Terminology, Storing Strings
2.2 State Character Data Type,
2.3 Discuss String Operations

STRING

A finite sequence S of **zero or more characters** is called a String.   Strings are actually one-dimensional array of characters terminated by a **null** character '\0'.

Empty string : The string with zero character is called the empty string or null string.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello".

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows –

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++ –

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

Program: 1

```
#include <stdio.h>
int main ()
{
  char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
  printf("Greeting message: %s\n", greeting );
  return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Greeting message: Hello

CHARACTER DATA TYPE:-

Character Data Types

Character data types are strings of characters. Upper and lower case alphabetic characters are accepted literally. There is

one fixed-length character data type: char   and

two variable-length character data types: varchar and long varchar.

Fixed-length char strings can contain any printing or non-printing character, and the null character

---

('\0'). Char strings are padded with blanks to the declared length.

Leading and embedded blanks are significant when comparing char strings.

For example, the following char strings are considered different:

"A B C"

"ABC"

Length is not significant when comparing char strings; the shorter string is (logically) padded to the length of the longer.

For example, the following char strings are considered equal:

"ABC"

"ABC "

varchar Data Types

Varchar strings are variable-length strings. The varchar data type can contain any character, including non-printing characters and the ASCII null character ('\0').

Except when comparing with char data, blanks are significant in the varchar data type.

For example, the following two varchar strings are not considered equal:

```
"the store is closed"
```

and

```
"thestoreisclosed"
```

If the strings being compared are unequal in length, the shorter string is padded with trailing blanks until it equals the length of the longer string.

For example, consider the following two strings:

```
'abcd\001'
```

where:

'\001' represents one ASCII character (ControlA)

and

```
'abcd'
```

The string data type is of two data type.

(1) Constant string

(2) Variable string

Constant String:

-> The constant string is fixed & is written in either ‗ ' single quote & ― ‖ double quotation.

Ex:- ‗SONA'

 ―Sona‖

Variable String:

String variable falls into 3 categories.

1. Static

2. Semi-Static

3. Dynamic

Static character variable:

Whose variable is defined before the program can be executed & cannot change through out the program.

Semi-static variable:

Whose length variable may as long as the length does not exist, a maximum

value. A maximum value determine by the program before the program is

executed.
Dynamic variable:
A variable whose length can change during the execution of the program.

String operations

C supports a wide range of functions that manipulate null-terminated strings –
1. SUBSTRING

 A substring is **a contiguous sequence of characters within a string**.
 For instance, "the best of" is a substring of "It was the best of times".
To access a substring we need the following information,

- **Name of the string.**

- **Position of the first character of the substring in the given string.**

- **Length of the substring.**

SUBSTRING (String, initial, length)
To denote the substring of string S beginning in the position K having a length L.
SUBSTRING (S, K, L)
For example, SUBSTRING ("TO BE OR NOT TO BE", 4, 7)
SUBSTRING=BE OR N
SUBSTRING ("THE END", 4, 4)
SUBSTRING= END.

2. Other functions on string

Sl. No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.

6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions –
Program 2
```
#include <stdio.h>
#include <string.h>
int main ()
{
char str1[12] = "Hello";
char str2[12] = "World";
char str3[12];
int len ;
  /* copy str1 into str3 */
strcpy(str3, str1);
printf("strcpy( str3, str1) :  %s\n", str3 );

  /* concatenates str1 and str2 */
strcat( str1, str2);
printf("strcat( str1, str2):   %s\n", str1 );

  /* total length of str1 after concatenation */
len = strlen(str1);
printf("strlen(str1) :  %d\n", len );
  return 0;
}
```
When the above code is compiled and executed, it produces the following result –
strcpy(str3, str1) : **Hello**
strcat(str1, str2):   **HelloWorld**
strlen(str1) : **10**

3.0 ARRAYS 07

3.1 Give Introduction about array,

3.2 Discuss Linear arrays, representation of linear array In memory

3.3 Explain traversing linear arrays, inserting & deleting elements

3.4 Discuss multidimensional arrays, representation of two dimensional arrays

in memory (row major order & column major order), and pointers

3.5 Explain sparse matrices.

3.1 Introduction about array

A Linear Array is a list of finite number of n homogeneous data elements i.e. the

elements of same data types Such that:

a) The elements of the array are referenced respectively by an index set

consisting of n consecutive numbers.

b) All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.
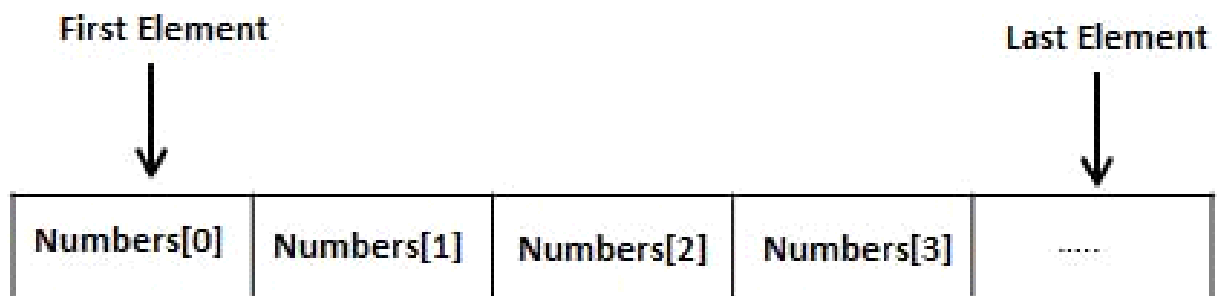
The number n of elements is called length or size of array. If not explicitly stated,

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

Representation of Linear Arrays in memory

The elements of linear array are stored in consecutive memory locations. It is shown below:
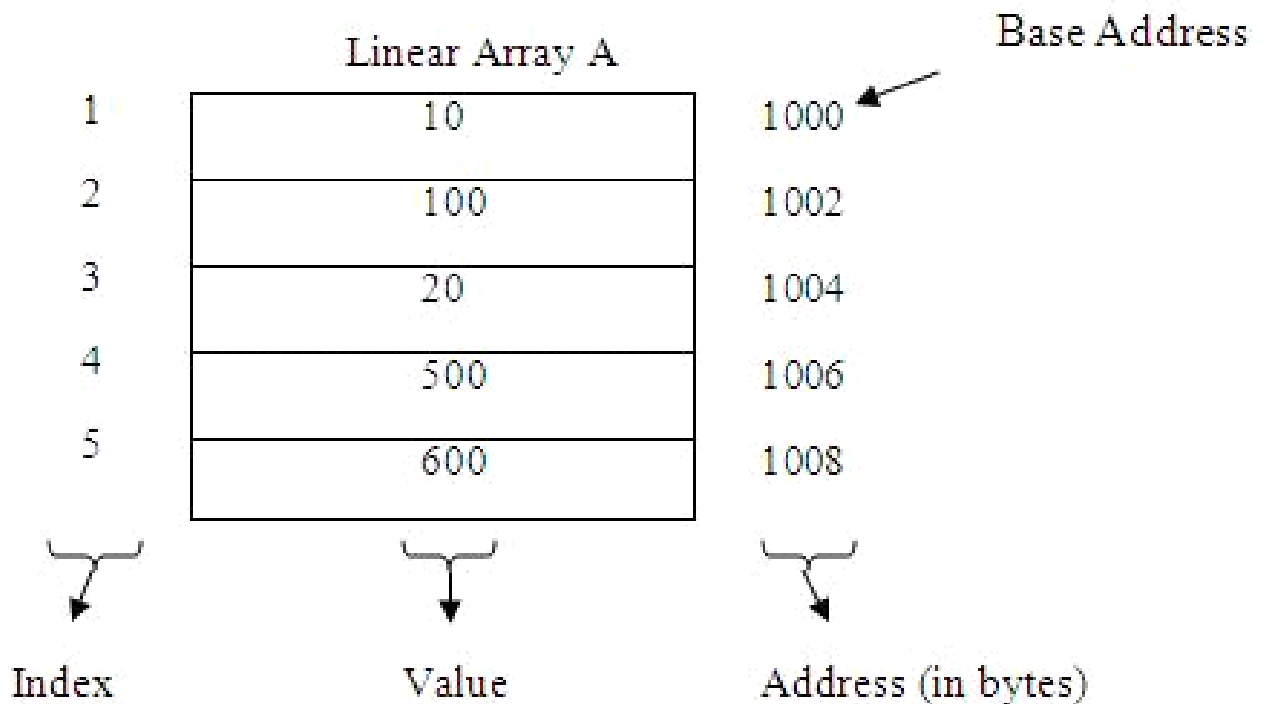
Let LA is a linear array in the memory of the computer. Recall that the memory

of computer is simply a sequence of addressed locations.

LOC (LA[k]) = address of element LA[k] of the array LA.

Let the address of the first element of LA, denoted byBase (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.

Following are the basic operations supported by an array.

- Traverse − Visit all the array elements one by one.

- Insertion − Adds an element at the given index.

- Deletion − Deletes an element at the given index.

- Search − Searches an element using the given index or by the value.

- Update − Updates an element at the given index.

Algorithm Of Traversal Of An Array

It is an operation in which element of the array is visited. The travel proceeds from first element to the last element of the array.

Algorithm :

1. C=1

2. Process LIST[C]

3. C= C+1

4. if (C<=N) then repeat 2 and 3

5. End.

Example:Following program traverses and prints the elements of an array:

```
#include <stdio.h>
main() {
  int LA[] = {1,3,5,7,8};
  int  n=5;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

When we compile and execute the above program, it produces the following result −

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Insertion Operation:-

- Insert operation is to insert one or more data elements into an array.

- Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Algorithm insertion

Let LA be a Linear Array (unordered) with N elements and K is a positive integer such that K<=N.

Following is the algorithm where ITEM is inserted into the Kth position of LA.

- Start

- Set J = N

- Set N = N+1

- Repeat steps 5 and 6 while J >= K

- Set LA[J+1] = LA[J]

- Set J = J-1

- Set LA[K] = ITEM

- Stop

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm : SEARCH

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start

2. Set J = 0

3. Repeat steps 4 and 5 while J < N

4. IF LA[J] is equal ITEM THEN GOTO STEP 6

5. Set J = J +1

6. PRINT J, ITEM

7. Stop

Example: Following is the implementation of the above algorithm —

```c
#include <stdio.h>
void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }


   while( j < n){
      if( LA[j] == item ) {
         break;
      }
      j = j + 1;
   }
   printf("Found element %d at position %d\n", item, j+1);
}
```

When we compile and execute the above program, it produces the following result —

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Found element 5 at position 3

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm : UPDATE

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm **to update an element available at the Kth position of LA.**

1. Start

2. Set LA[K-1] = ITEM

3. Stop

Example : Following is the implementation of the above algorithm —

```
#include <stdio.h>
void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5, item = 10;
   int i, j;
      printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
      LA[k-1] = item;
   printf("The array elements after updation :\n");


   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result —

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after updation :

LA[0] = 1

LA[1] = 3

LA[2] = 10

LA[3] = 7

LA[4] = 8

Multidimensional Array

1. Array having more than one subscript variable is called Mult iDimensional 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

faceprep

Two-dimensional Array
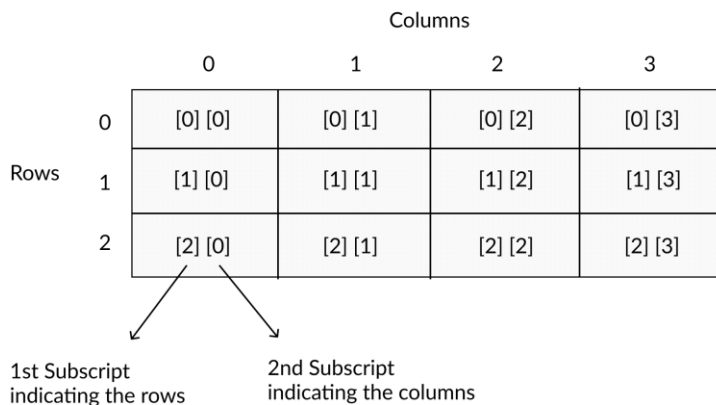
Memory Representation

1. 2-D arrays are Stored in contiguous memory location row wise.

2. 3 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 mem. locations.

6. This is integer array so each element requires 2 bytes of memory.

Basic Memory Address Calculation :

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

Element Memory Location

Element	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 Representation:

- Column-major
- Row-major

For example,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

Address	Row-major order	Column-major order
0	a_{11}	a_{11}
1	a_{12}	a_{21}

2	a_{13}	a_{12}
3	a_{21}	a_{22}
4	a_{22}	a_{13}
5	a_{23}	a_{23}

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

What is Sparse Matrix?

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represent a m X n matrix. There may be a situation in which a matrix

contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times. To make it simple we use the following sparse matrix representation.

Sparse Matrix Representations: Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...
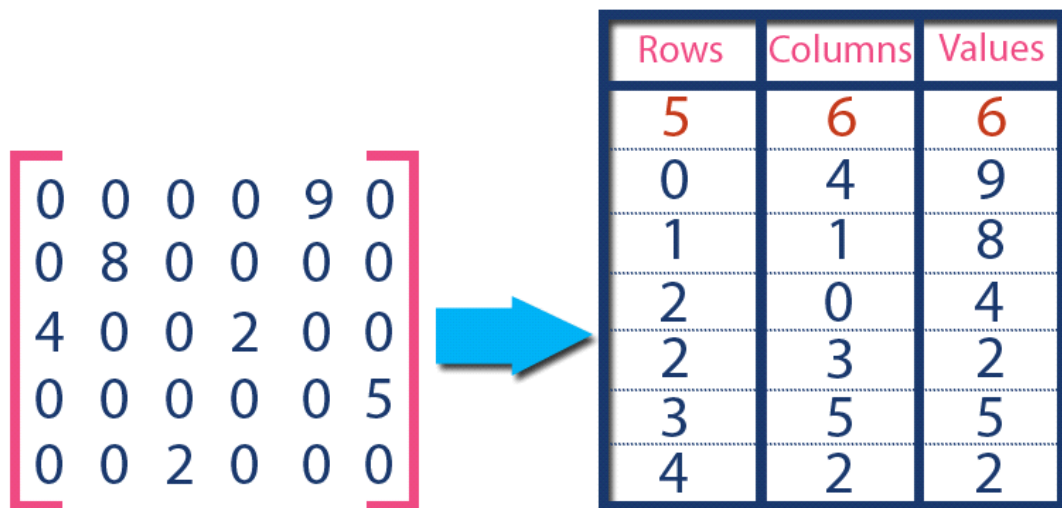
Sparse Matrix Triplet Representation

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow a similar pattern.

Implementation of Array Representation of Sparse Matrix using C++

#include<iostream>

```cpp
using namespace std;

int main()
{
    // sparse matrix of class 5x6 with 6 non-zero values
    int sparseMatrix[5][6] =
    {
        {0 , 0 , 0 , 0 , 9, 0 },
        {0 , 8 , 0 , 0 , 0, 0 },
        {4 , 0 , 0 , 2 , 0, 0 },
        {0 , 0 , 0 , 0 , 0, 5 },
        {0 , 0 , 2 , 0 , 0, 0 }
    };


    // Finding total non-zero values in the sparse matrix
    int size = 0;
    for (int row = 0; row < 5; row++)
        for (int column = 0; column < 6; column++)
            if (sparseMatrix[row][column] != 0)
                size++;

    // Defining result Matrix
    int resultMatrix[3][size];

    // Generating result matrix
    int k = 0;
    for (int row = 0; row < 5; row++)
        for (int column = 0; column < 6; column++)
            if (sparseMatrix[row][column] != 0)
            {
                resultMatrix[0][k] = row;
                resultMatrix[1][k] = column;
                resultMatrix[2][k] = sparseMatrix[row][column];
                k++;
            }

    // Displaying result matrix
    cout<<"Triplet Representation : "<<endl;
    for (int row=0; row<3; row++)
    {
        for (int column = 0; column<size; column++)
            cout<<resultMatrix[row][column]<<" ";

        cout<<endl;
    }
    return 0;
}
```

ASSIGNMENT: 3

Q.1   What is an array. how it can be represented in memory?

Q.2   What are the basic operations performed in an array.

Q.3   write algorithm to insert an element in a specific location.

Q.4   Write algorithm to delete a particular element from an array.

Q.5   calculate the address of element A [1, 2]of the matrix is of 2*4. given base_address = 1000

Q.6   explain row major and column major representation of a 2d array.

Q.7   What is sparse matrx and how it can be reprsented in memory.

programming assignment: in c/c++

1. Read a list of 10 elements and print the same.

2. insrt an element 100 at location 3 in an array of 5 elements.

3. delete the element at location 4 in an array of 6 elements.

4. you have wrongly entered a value 5 instead of 55. correct the mistake.

5. search an elemnt 25 in a given array. find its location

6. implement a spase matrix of 5*5 with three non zero elements.

4.0 STACKS & QUEUES   08
4.1 Give fundamental idea about Stacks and queues
4.2 Explain array representation of Stack
4.3 Explain arithmetic expression ,polish notation & Conversion
4.4 Discuss application of stack, recursion
4.5 Discuss queues, circular queue, priority queues

Stacks:A stack is a container of objects that are inserted and removed at one end known as top. It follows last-in first-out (LIFO) principle.
A stack is a restricted data structure, because only a small number of operations are performed on it.

1 . **push** the item into the stack,

2.   **pop** the item out of the stack.

A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

A stack is a **recursive** data structure. Here is a structural definition of a Stack:

a stack is either empty or

it consistes of a top and the rest which is a stack;
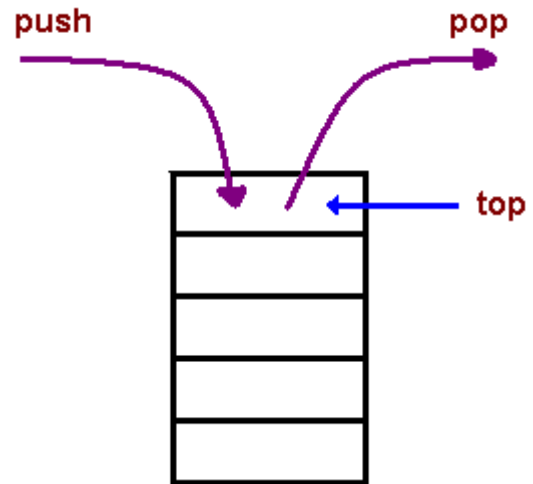
Applications

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.

ARRAY Representation of STACK

Stack may be represented in computer in various ways(by means of Link List or linear array).

Unless otherwise stated or implied, stack will be maintained by means of

i. Linear Array STACK.

ii. A pointer variable TOP which contains the location of top element Of the stack.

iii. A variable MAXSTK which gives the maximum number of elements that can be held by the stack.

Condition TOP=0 will indicate that the stack is empty and TOP=MAXSTK indicates that the stack is full

In a **dynamic stack abstraction** when *top* reaches *capacity*, we double up the stack size.
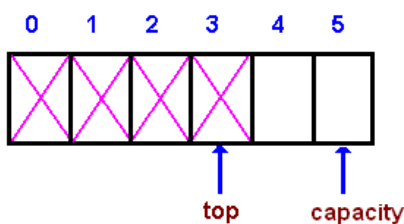
Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.
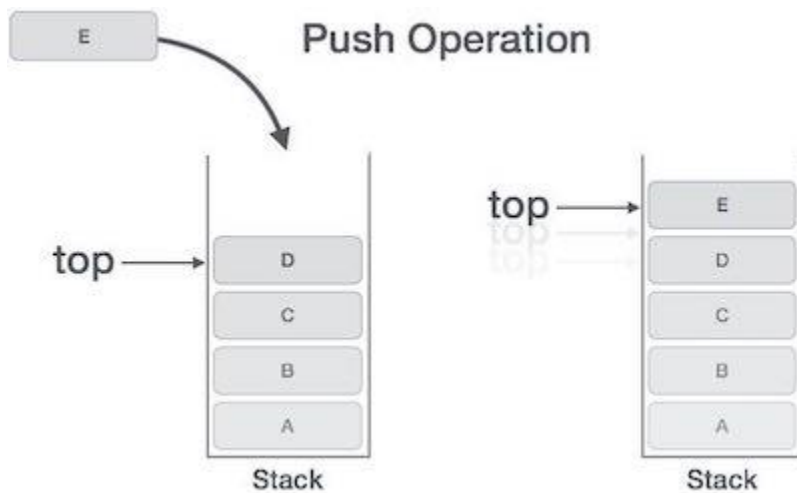
Push Operation

Algorithm for PUSH Operation

Algorithm

PUSH(STACK,TOP,MAXSTK, ITEM)

    1. If TOP=MAXSTK   [Stack already filled]

         Then print: OVERFLOW and Return.
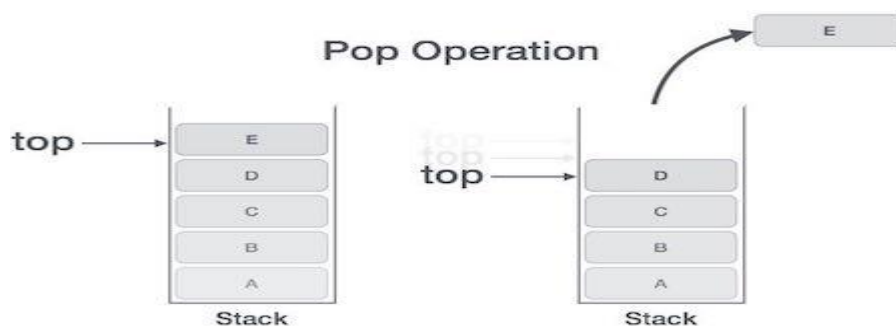
    2. TOP=TOP+1

    3. STACK[TOP]=ITEM   [insert ITEM in new TOP position]

    4. Return.

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.

- **Step 2** – If the stack is empty, produces an error and exit.

- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.

- **Step 4** – Decreases the value of top by 1.

- **Step 5** – Returns success.

Pop Operation

Algorithm for Pop Operation:

Algorithm

POP(STACK,TOP, ITEM)

    1. If TOP=0   [Stack has no ITEM to be removed]

Then print: UNDERFLOW and Return.
2. ITEM=STACK[TOP]   [Assign TOP element to ITEM]
3. TOP=TOP-1   [Decreases TOP by 1]
4. Return.

What are some downsides to using a stack?
• No random access. You get the top, or nothing.
• No walking through the stack at all — you can only reach an element by popping all the elements higher up off first
• No searching through a stack.

What are some benefits to using a stack?
• Useful for lots of problems -- many real-world problems can be solved with a LastIn-First-Out model (we'll see one in a minute)
• Very easy to build one from an array such that access is guaranteed to be fast.
• Where would you have the top of the stack if you built one using a Vector? Why would that be fast?

Arithmetic expression, polish notation & Conversion
The way to write arithmetic expression is known as a **notation**. **Polish Notation** is a general form of expressing mathematical, logical and algebraic equations. The compiler uses this notation in order to evaluate mathematical expressions depending on the order of operations. There are in general three types of Notations used while parsing Mathematical expressions:

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation
We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation
In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation
This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.
The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +

| 5 | (a + b) ∗ (c + d) | ∗ + a b + c d | a b + c d + ∗ |
| 6 | ((a + b) ∗ c) - d | - ∗ + a b c d | a b + c ∗ d - |

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example –

a + b * c ⟹ a + (b * c)

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and − have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example –

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Algorithm : infix to postfix

Let 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 to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then :
(a) Add ⊗ to STACK.
[End of If structure].
(b) Repeatedly pop from STACK and add P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
6. If a right parenthesis is encountered, then :
(a) Repeatedly pop from STACK and add to P each operator (on the top of STACK until a

left parenthesis is encountered.
(b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
[End of if structure.]
 [End of Step 2 loop].
7. Exit

Examples for converting infix expression to postfix form :

1) Give postfix form for A + B − C
Sol.
 (A + B) − C
 (AB +) − C
Let T = (AB +)
 T − C
 TC −
or AB + C − Postfix expression

2. Give postfix form for A * B + C
Sol.
 (A * B) + C
 (AB *) + C
Let T = (AB *)
 T + C
 TC +
or AB * C + Postfix expression

3. Give postfix form for A * B + C/D
Sol.
 (A * B) + C/D
 (AB *) + C/D
Let T = (AB *)
 T + C/D
 T + (C/D)
 T + (CD /)
Let S = (CD /)
 T + S
 TS +
or AB * CD / + Postfix expression

4. Give postfix form for A + B/C − D

Sol.
 A + (B/C) − D
 A + (BC/) − D
Let T = (BC /)
 A + T − D
 (A + T) − D
 (AT +) − D
Let S = (AT +)
 S − D
 SD −
 AT + D −
 ABC / + D − Postfix expression

5. Give postfix form for (A + B)/(C − D)
Sol.
 (A + B)/(C − D)

(AB +)/(C − D)

(AB +)/(CD −)

Let T = (AB +) & S = (CD −)

T/S

TS /

AB + CD − / Postfix expression

6. Give postfix form for (A + B) * C/D

Sol.

(A + B) * C/D

(AB +) * C/D

Let T = (AB +)

T * C/D

(T * C)/D

(TC *)/D

Let S = (TC *)

S/D

SD /

TC * D /

AB + C * D/ Postfix expression

7. Give postfix form for (A + B) * C/D + E ^ F/G

Sol.

(AB +) * C/D + E ^ F / G

Let T = (AB +)

T * C/D + (E^F) / G

T * C/D + (EF ^) / G

Let S = (EF ^)

T * C/D + S/G

(T * C)/D + S/G

(TC *)/D + S/G

Let Q = (TC *)

Q/D + S/G

(Q/D) + S/G

(QD /) + S/G

Let P = (QD /)

P + S/G

P + (S/G)

P + (SG /)

Let O = (SG /)

P + O

PO +

Now we will expand the expression PO +

PO +

PSG/ +

QD / SG / +

TC * D / SG / +

TC * D / EF ^ G / +

AB + C * D / EF ^ G / + Postfix expression

8. Give postfix form for A + [(B + C) + (D + E) * F]/G

Sol.

A + [(B + C) + (D + E) * F]/G

A + [(BC +) + (DE +) * F] / G

Let T = (BC +) & S = (DE +)

A + [T + S * F] / G

A + [T + (SF *)] / G

Let Q = (SF *)

A + [T + Q] / G

A + (TQ +) / G

Let P = (TQ +)

A + P / G

A + (PG /)

Let N = (PG /)

A + N

AN +

Expanding the expression AN + gives

APG / +

ATQ + G / +

ATSF * + G / +

ABC + DE + F * + G / + Postfix expression

9. **Give postfix form for A + (B * C – (D / E ^ F) * G) * H.**

Sol.

A + (B * C – (D / E ^ F) * G) * H

A + (B * C – (D / (EF ^)) * G) * H

Let T = (EF ^)

A + (B * C – (D / T) * G) * H

A + (B * C – (DT /) * G) * H

Let S = (DT /)

A + (B * C – S * G) * H

A + (B * C – (SG *)) * H

Let Q = (SG *)

A + (B * C – Q) * H

A + ((B * C) – Q) * H

A + ((BC *) – Q) * H

Let P = (BC *)

A + (P – Q) * H

A + (PQ –) * H

Let O = (PQ –)

A + O * H

A + (OH *)

Let N = (OH *)

A + N

AN +

Expanding the expression AN + gives,

AOH * +

APQ – H * +

ABC * Q – H * +

ABC * SG * – H * +

ABC * DT / G * – H * +

ABC * DEF ^ / G * – H * + Postfix expression

10. **Give postfix form for A – B / (C * D ^ E).**

Sol.

A – B / (C * D ^ E)

A – B / (C * (DE ^))

Let T = (DE ^)
A – B / (C * T)
A – B / (CT *)
Let S = (CT *)
A – B / S
A – (BS /)
Let Q = (BS /)
A – Q
AQ –
Now expanding the expression AQ –
AQ –
ABS / –
ABCT * / –
ABCDE ^ * / – Postfix exp

Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W

/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

The final postfix expression of infix expression(K + L - M*N + (O^P) * W/U/V * T + Q) is KL+MN*-OP^W*U/V/T*+Q+.

Discuss application of stack, recursion

Following is the various Applications of Stack in Data Structure:

- o Evaluation of Arithmetic Expressions
- o Backtracking
- o Delimiter Checking
- o Reverse a Data
- o Processing Function Calls

Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Let us consider a problem that a programmer have to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply add the numbers starting from 1 to n. So the function simply looks like,

approach(1) – Simply adding one by one

f(n) = 1 + 2 + 3 +……..+ n

but there is another mathematical approach of representing this,

approach(2) – Recursive adding

f(n) = 1   n=1

f(n) = n + f(n-1)   n>1

There is a simple difference between the approach (1) and approach(2) and that is in **approach(2)** the function " **f( )** " itself is being called inside the function, so this phenomenon is named as recursion and the function containing recursion is called recursive function, at the end this is a great tool in the hand of the programmers to code some problems in a lot easier and efficient way.

What is base condition in recursion?
In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
   if (n < = 1) // base case
      return 1;
   else
      return n*fact(n-1);
}
```

In the above example, **base case for n < = 1 is defined** and larger value of number can be solved by converting to smaller one till base case is reached.

Example: Calculate Factorial Using Recursion

```
#include<iostream>

int factorial(int n);

int main()
{
   int n;

cout<< "Enter a positive integer: ";
cin>> n;

cout<< "Factorial of " << n << " = " << factorial(n);

   return 0;
}

 Long int factorial(int n)
{
```

```
if(n > 1)
    return n * factorial(n - 1);
  else
    return 1;
}
```

Output:
Enter an positive integer: 6
Factorial of 6 = 720

--end of stack--------------------------------

4.5 Discuss queues, circular queue, priority queues
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (**rear end)** and the other is used to remove data (**front end**). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Representation of Queue:
Queue may be represented in computer in various ways(by means of Link List or linear array). Unless otherwise stated or implied, queue will be maintained by means of Linear Array QUEUE.Two pointer variables FRONT which contains the location offrontelement of the Queue & REAR, containing the location of last element.
i.   Condition FRONT=0 will indicate that the Queue is empty
ii.   whenever an element is deleted from Queue, the value of FRONT is increased by 1, i.e.FRONT=FRONT+1
iii.   Similarly, whenever an element is added to the Queue, the value of REAR is also increased by 1, i.e. REAR=REAR+1

Algorithm for insertion into the Queue
QINSERT(QUEUE,N,FRONT,REAR,ITEM)
   1. If FRONT=1 and REAR=N or FRONT=REAR+1
            then write OVERFLOW and Return. [Queue already filled]
   2. If FRONT=0   [Queue initially empty]
            then FRONT=1 and REAR=1.
            else if REAR=N
                      then REAR=1.
                       else REAR=REAR+1.

3. QUEUE[REAR]=ITEM.
4. Exit.

Algorithm for deletion operation in the Queue
QDELETE(QUEUE,N,FRONT,REAR,ITEM)
1. If FRONT=0

then write UNDERFLOW and Return. [Queue already empty]
2. ITEM=QUEUE[FRONT]
3. If FRONT=REAR   [Queue has only one element]

then FRONT=0 and REAR=0.

else if FRONT=N

then FRONT=1.

 else FRONT=FRONT+1.
3. Exit.

circular queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as **Ring Buffer** as all the ends are connected to another end. The circular queue can be represented as:
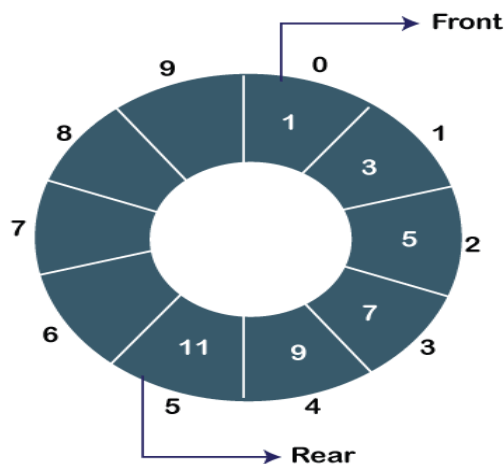
The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.
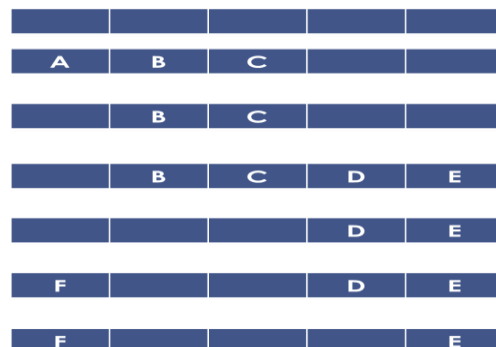
Applications of Circular Queue
The circular Queue can be used in the following scenarios:
  o **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of

a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.

- o **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.

- o **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Priority Queue

A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.

In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:

The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

Deque (Double Ended Queue)

In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.

Assignment:

1.

2. Give postfix form for A – B / (C * D ^ E).
3. Give postfix form for A + B/C – D
4. Give postfix form for (A + B) * C/D + E ^ F/G
5. Give postfix form for A + [(B + C) + (D + E) * F]/G
6. Give postfix form for A + (B * C – (D / E ^ F) * G) * H.
7. Write push and pop algorithm for stack.
8. Write Q insertion and Q deletion   algorithm.
9. What is circular queue. How its different from queue.
10. What is priority queue. Where its used.
11. Name different types of queues.

Programming assignment:

1. */*program 1*

2. ** C Program to Implement a Queue using an Array*

```c
3.   */
4.   #include <stdio.h>
5.
6.   #define MAX 50
7.
8.   void insert();
9.   void delete();
10.  void display();
11.  int queue_array[MAX];
12.  int rear = - 1;
13.  int front = - 1;
14.  main()
15.  {
16.     int choice;
17.     while (1)
18.     {
19.  printf("1.Insert element to queue \n");
20.  printf("2.Delete element from queue \n");
21.  printf("3.Display all elements of queue \n");
22.  printf("4.Quit \n");
23.  printf("Enter your choice : ");
24.  scanf("%d", &choice);
25.        switch (choice)
26.        {
27.           case 1:
28.  insert();
29.  break;
30.           case 2:
31.  delete();
32.  break;
33.           case 3:
```

```
34. display();

35. break;

36.      case 4:

37. exit(1);

38.      default:

39. printf("Wrong choice \n");

40.    } /* End of switch */

41.  } /* End of while */

42. } /* End of main() */

43.

44. void insert()

45. {

46.   int add_item;

47.   if (rear == MAX - 1)

48. printf("Queue Overflow \n");

49.   else

50.   {

51.     if (front == - 1)

52. /*If queue is initially empty */

53.     front = 0;

54. printf("Inset the element in queue : ");

55. scanf("%d", &add_item);

56.     rear = rear + 1;

57. queue_array[rear] = add_item;

58.   }

59. } /* End of insert() */

60.

61. void delete()

62. {

63.   if (front == - 1 || front > rear)

64.   {
```

```
65.   printf("Queue Underflow \n");

66.   return ;

67.   }

68.   else

69.   {

70.   printf("Element deleted from queue is : %d\n", queue_array[front]);

71.       front = front + 1;

72.   }

73. } /* End of delete() */

74.

75. void display()

76. {

77.   int i;

78.   if (front == - 1)

79. printf("Queue is empty \n");

80.   else

81.   {

82. printf("Queue is : \n");

83.       for (i = front; i<= rear; i++)

84. printf("%d ", queue_array[i]);

85. printf("\n");

86.   }

87. } /* End of display() */
```

Program Explanation

1. Ask the user for the operation like insert, delete, display and exit.
2. According to the option entered, access its respective function using switch statement. Use the variables front and rear to represent the first and last element of the queue.
3. In the function insert(), firstly check if the queue is full. If it is, then print the output as "Queue Overflow". Otherwise take the number to be inserted as input and store it in the variable add_item. Copy the variable add_item to the array queue_array[] and increment the variable rear by 1.
4. In the function delete(), firstly check if the queue is empty. If it is, then print the output as "Queue Underflow". Otherwise print the first element of the array queue_array[] and decrement the variable front by 1.
5. In the function display(), using for loop print all the elements of the array starting from front to rear.
6. Exit.

Runtime Test Cases
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 15
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 20
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
15 20 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 4

Program 2: **Reversing the words in a sentence// Simple Stack Example**

```
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;
const char SPACE = ' ';
int main() {
```

```cpp
 string sentence = "hope is what defines humanity";
 string word;
 Stack<string>wordStack;
cout<< "Original sentence: " << sentence <<endl;
 for (char c : sentence) {
 if (c == SPACE) {
wordStack.push(word);
 word = ""; // reset
 } else {
 word += c;
 }
 }
 if (word != "") {
wordStack.push(word);
 }
cout<< " New sentence: ";
 while (!wordStack.isEmpty()) {
 word = wordStack.pop();
cout<< word << SPACE;
 }
cout<<endl;
 return 0;
}
```

program 3: menu driven program for stack
```c
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

printf("Enter the number of elements in the stack ");
scanf("%d",&n);
printf("*********Stack operations using array********");

printf("\n-------------------------------------------\n");
while(choice != 4)
   {
printf("Chose one from the below options...\n");
printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
printf("\n Enter your choice \n");
scanf("%d",&choice);
     switch(choice)
     {
       case 1:
       {
push();
        break;
       }
       case 2:
```

```c
                {
pop();
            break;
        }
        case 3:
        {
show();
            break;
        }
        case 4:
        {
printf("Exiting....");
            break;
        }
        default:
        {
printf("Please Enter valid choice ");
        }
    };
    }
}

void push ()
{
    int val;
    if (top == n )
printf("\n Overflow");
    else
    {
printf("Enter the value?");
scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
{
if(top == -1)
printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
printf("%d\n",stack[i]);
    }
if(top == -1)
    {
```

```c
        printf("Stack is empty");
    }
}
```

Program 4 program on recursion

4.Write a program in C to print first 50 natural numbers using recursion.

```c
#include<stdio.h>
int  numPrint(int);
int main()
{
   int n = 1;
        printf("\n\n Recursion : print first 50 natural numbers :\n");
        printf("-----------------------------------------------\n");
        printf(" The natural numbers are :");
numPrint(n);
printf("\n\n");
   return 0;
}
int numPrint(int n)
{
   if(n<=50)
   {
printf(" %d ",n);
numPrint(n+1);
   }
}
```

   4.   Write a program in C to Print Fibonacci Series using recursion.

C Code:

```c
#include<stdio.h>

int term;
int fibonacci(int prNo, int num);

void main()
{
  static int prNo = 0, num = 1;
        printf("\n\n Recursion : Print Fibonacci Series :\n");
        printf("--------------------------------------\n");

printf(" Input number of terms for the Series (< 20) : ");
scanf("%d", &term);
printf(" The Series are :\n");
printf(" 1  ");
fibonacci(prNo, num);
printf("\n\n");
}

int fibonacci(int prNo, int num)
{
  static int i = 1;
  int nxtNo;
```

```c
  if (i == term)
    return (0);
  else
  {
nxtNo = prNo + num;
prNo = num;
num = nxtNo;
printf("%d  ", nxtNo);

i++;
fibonacci(prNo, num); //recursion, calling the function fibonacci itself
  }
  return (0);
}
```

Sample Output:

Recursion : Print Fibonacci Series :

 Input number of terms for the Series (< 20) : 10
 The Series are :
1 1 2 3 5 8 13 21 34 55

5. **Calculate the factorial of a number using recursion.**
6. **Write a program in C to find the sum of digits of a number using recursion.**
7. **Write a program in C to convert a decimal number to binary using recursion**

```c
#include<stdio.h>
long convertBinary(int);
int main()
{
  long biNo;
  int decNo;

        printf("\n\n Recursion : Convert decimal number to binary :\n");
        printf("-------------------------------------------------\n");

printf(" Input any decimal number : ");
scanf("%d",&decNo);

biNo = convertBinary(decNo);//call the function convertBinary
printf(" The Binary value of decimal no. %d is : %ld\n\n",decNo,biNo);
  return 0;
}
long convertBinary(int decNo)
{
  static long biNo,r,fctor = 1;

if(decNo != 0)
  {
      r = decNo % 2;
biNo = biNo + r * fctor;
```

```
fctor = fctor * 10;
convertBinary(decNo / 2);//calling the function convertBinary itself recursively
    }
    return biNo;
}
```

Sample Output:

Recursion : Convert decimal number to binary :

 Input any decimal number : 66
 The Binary value of decimal no. 66 is : 1000010
Program 8:Write a program in C to print even or odd numbers in given range using recursion
Program :9Write a program in C to find the LCM of two numbers using recursion
 Program :10Write a program in C to check a number is a prime number or not using recursion.

ADDRESS	INFO	LINK
101	I	112
102	C	110
103	U	111
104	P	103
105		
106	M	104
107	U	113
108	E	109
109	R	114
110	O	106
111	T	108
112	R	107
113	S	0
114		115
115	V	101

102

6.0 TREE 08
6.1 Explain Basic terminology of Tree
6.2 Discuss Binary tree, its representation and traversal, binary search tree,searching,
6.3 Explain insertion & deletion in a binary search trees
6.1 Explain Basic terminology of Tree
A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that **each node of the tree stores a value, a list of references to nodes (the "children").**
Recursive Definition: A tree consists of a root, and zero or more subtrees T_1, T_2, ... , T_k such that there is an edge from the root of the tree to the root of each subtree.
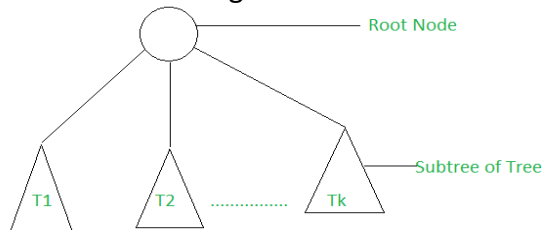
Basic Terminology In Tree Data Structure:
A tree T is represented by nodes and edges, which includes:
1.   **T** is empty (called null or empty tree).

2.   **T** has a left subtree and right subtree.

1. Root-
 The first node from where the tree originates is called as a **root node**.
 *   In any tree, there must be only one root node.

 *   We can never have multiple root nodes in a tree data structure.

 Example-
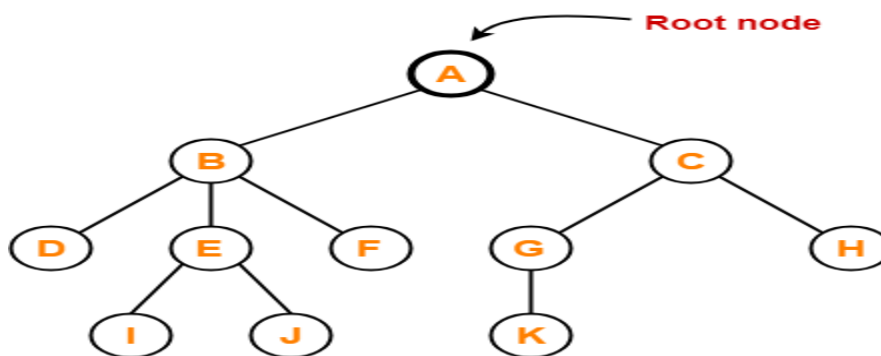
Here, node A is the only root node.
 2. Edge-
 *   The connecting link between any two nodes is called as an **edge**.

 *   In a tree with n number of nodes, there are exactly (n-1) number of edges.

 Example-
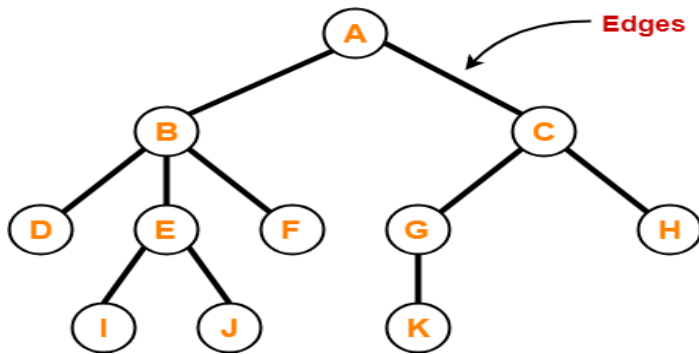
3. Parent-

The node which has a branch from it to any other node is called as a **parent node**.

- In other words, the node which has one or more children is called as a parent node.

- In a tree, a parent node can have any number of child nodes.

Example-
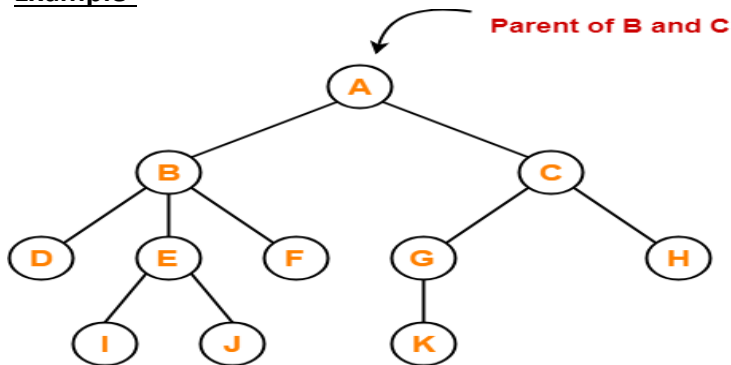
Here,

- Node A is the parent of nodes B and C

- Node B is the parent of nodes D, E and F

- Node C is the parent of nodes G and H

- Node E is the parent of nodes I and J

- Node G is the parent of node K

4. Child-

The node which is a descendant of some node is called as a **child node**.

- All the nodes except root node are child nodes.

Example-
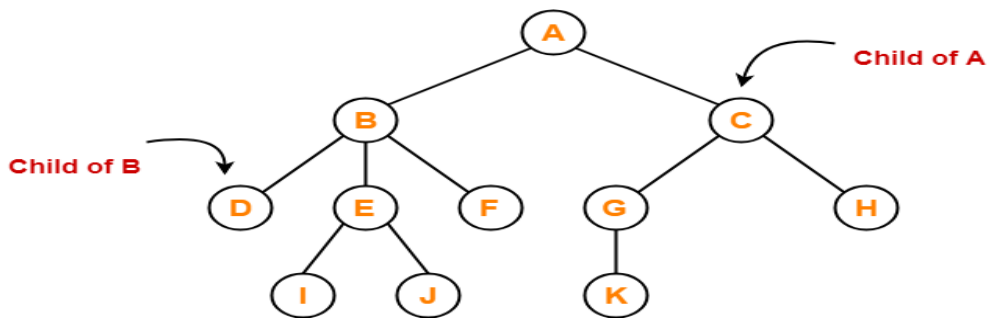
Here,

- Nodes B and C are the children of node A

- Nodes D, E and F are the children of node B

- Nodes G and H are the children of node C

- Nodes I and J are the children of node E

- Node K is the child of node G

5. Siblings-

Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.

Example-
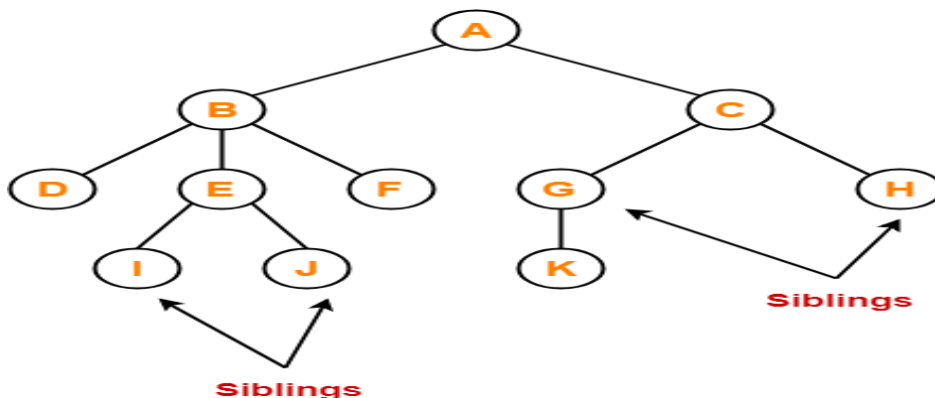
Here,

- Nodes B and C are siblings

- Nodes D, E and F are siblings

- Nodes G and H are siblings

- Nodes I and J are siblings

6. Degree-

Degree of a node is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Example-
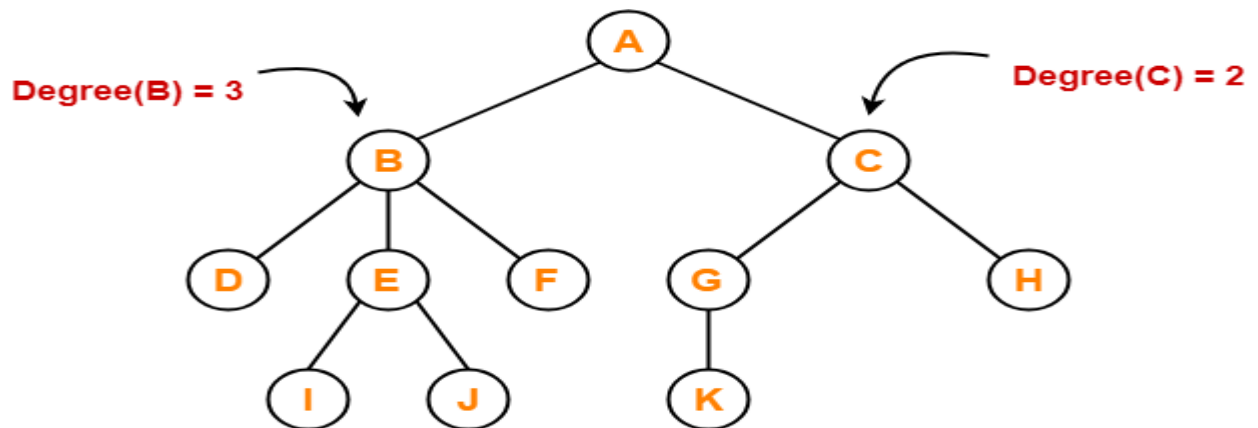
Here,

- Degree of node A = 2

- Degree of node B = 3

- Degree of node C = 2

- Degree of node D = 0

- Degree of node E = 2

- Degree of node F = 0

- Degree of node G = 1

- Degree of node H = 0

- Degree of node I = 0

- Degree of node J = 0

- Degree of node K = 0

7. Internal Node-
The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.

- Every non-leaf node is an internal node.

Example-
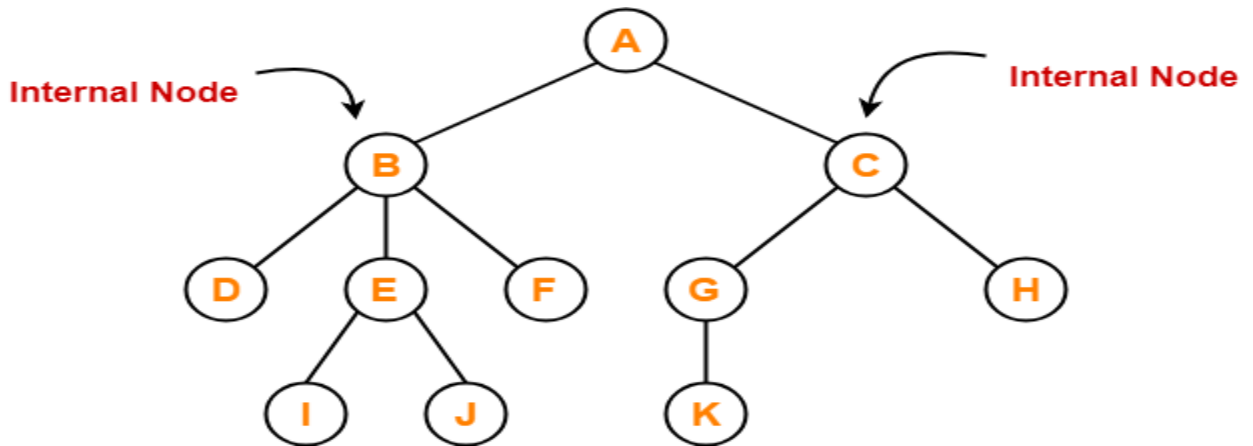
Here, nodes A, B, C, E and G are internal nodes.

8. Leaf Node-

The node which does not have any child is called as a **leaf node**.

- Leaf nodes are also called as **external nodes** or **terminal nodes**.

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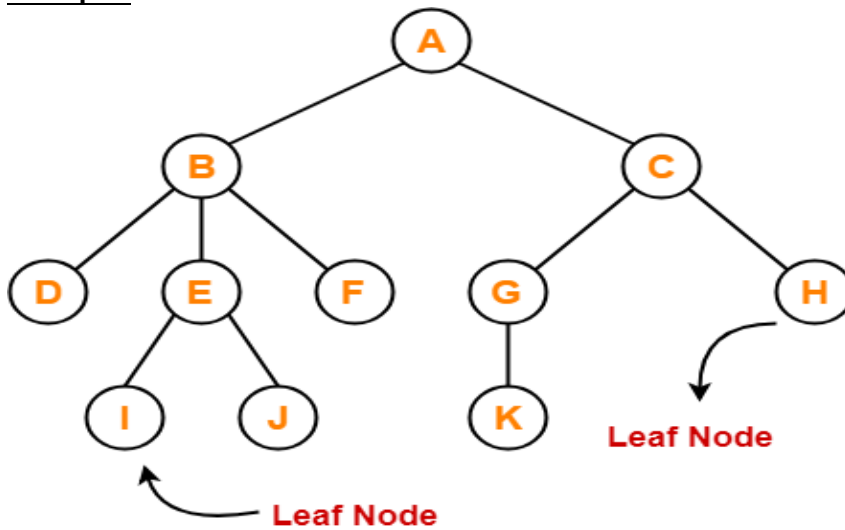
Here, nodes D, I, J, F, K and H are leaf nodes.

9. Level-

In a tree, each step from top to bottom is called as **level of a tree**.

- The level count starts with 0 and increments by 1 at each level or step.

Example-
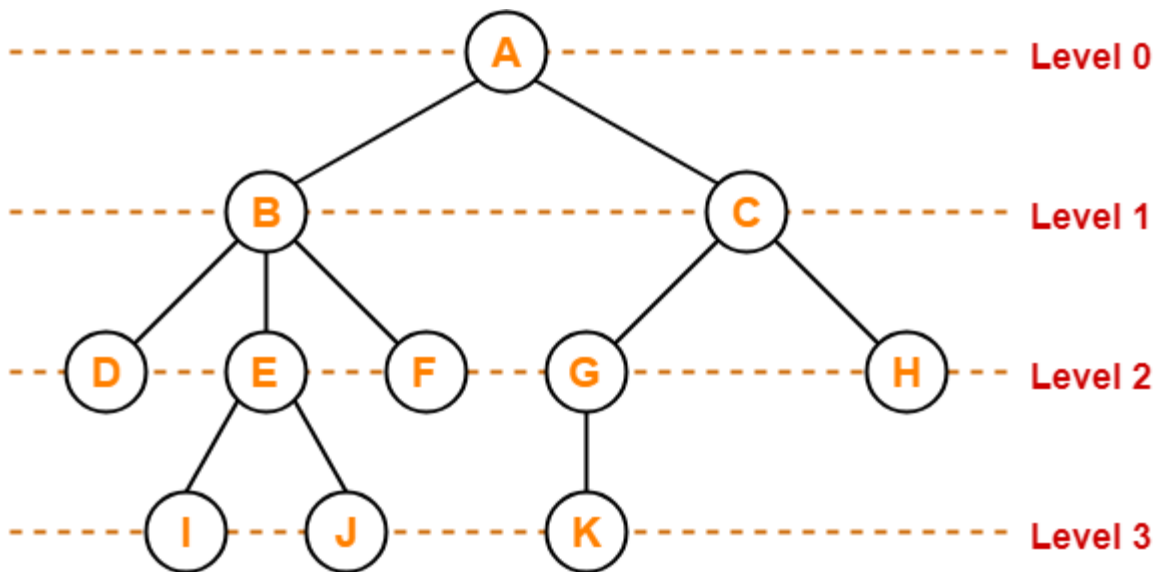
10. Height-

Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.

- **Height of a tree** is the height of root node.

- Height of all leaf nodes = 0

Example-
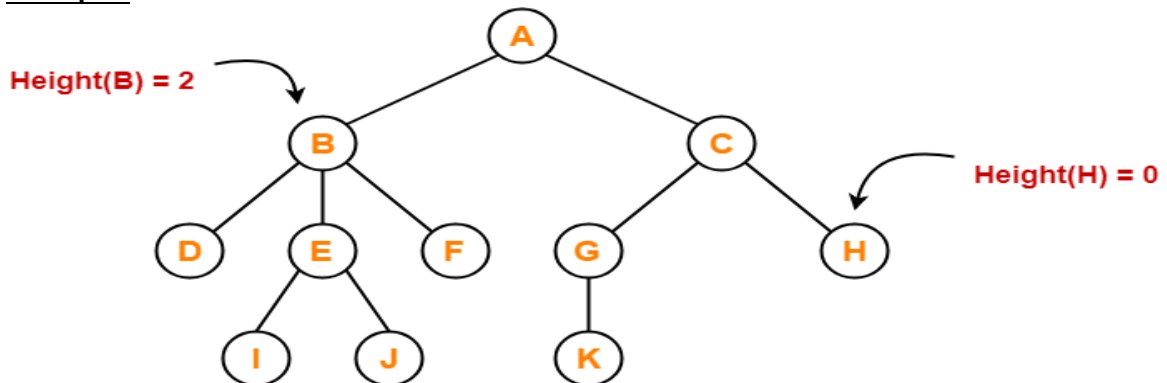
Height(B) = 2

Height(H) = 0

Here,

- Height of node A = 3

- Height of node B = 2

- Height of node C = 2

- Height of node D = 0

- Height of node E = 1

- Height of node F = 0

- Height of node G = 1

- Height of node H = 0

- Height of node I = 0

- Height of node J = 0

- Height of node K = 0

11. Depth-

Total number of edges from root node to a particular node is called as **depth of that node**.

- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.

- Depth of the root node = 0

- The terms "level" and "depth" are used interchangeably.

Example-
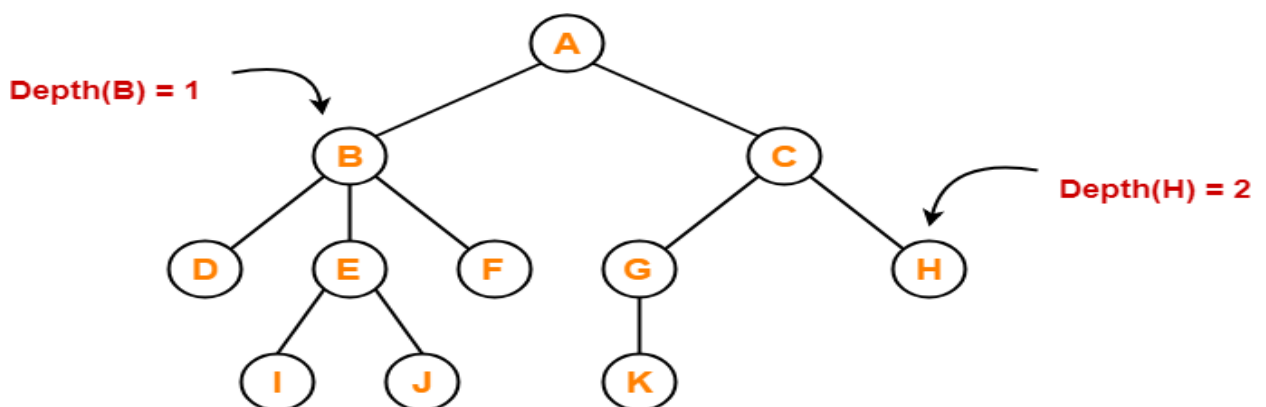
Here,

- Depth of node A = 0

- Depth of node B = 1

- Depth of node C = 1

- Depth of node D = 2

- Depth of node E = 2

- Depth of node F = 2

- Depth of node G = 2

- Depth of node H = 2

- Depth of node I = 3

- Depth of node J = 3

- Depth of node K = 3

12. Subtree-

In a tree, each child from a node forms a **subtree** recursively.

- Every child node forms a subtree on its parent node.

Example-

Sub trees
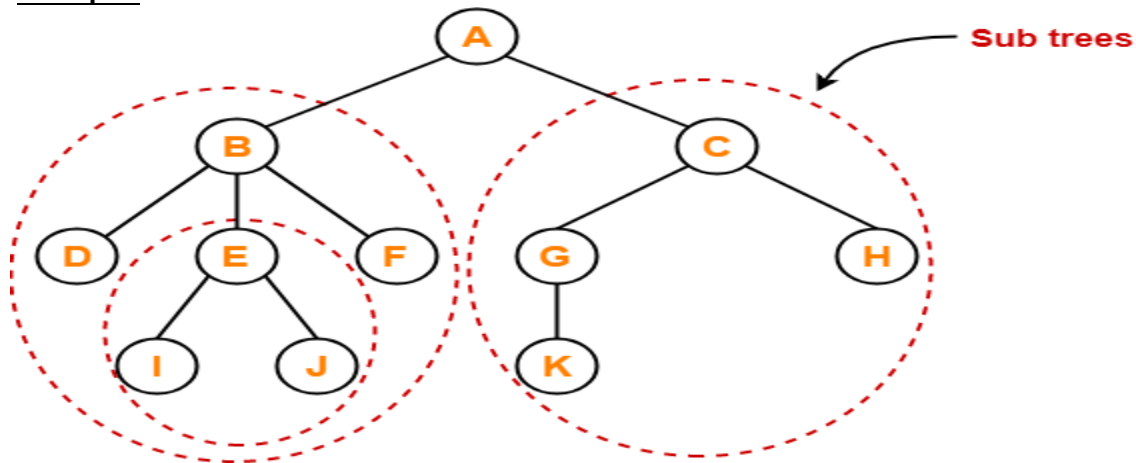
13. Forest-
A forest is a set of disjoint trees.
Example-

Forest
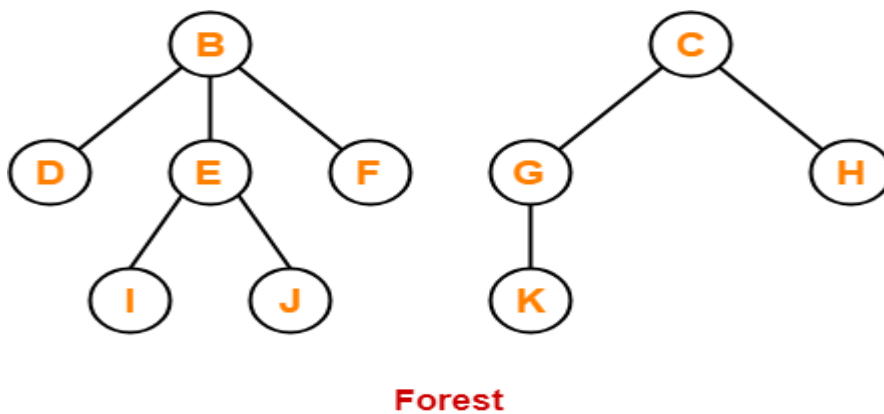
6.2 Discuss Binary tree, its representation and traversal, binary search tree,searching,
Discuss Binary tree
Binary tree is a special tree data structure in which each node can have at most 2 children.
Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children.Since each element in a binary tree can have only 2 children, we typically name them the **left and right child**.
A Binary Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child

Full Binary Tree A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

Example-
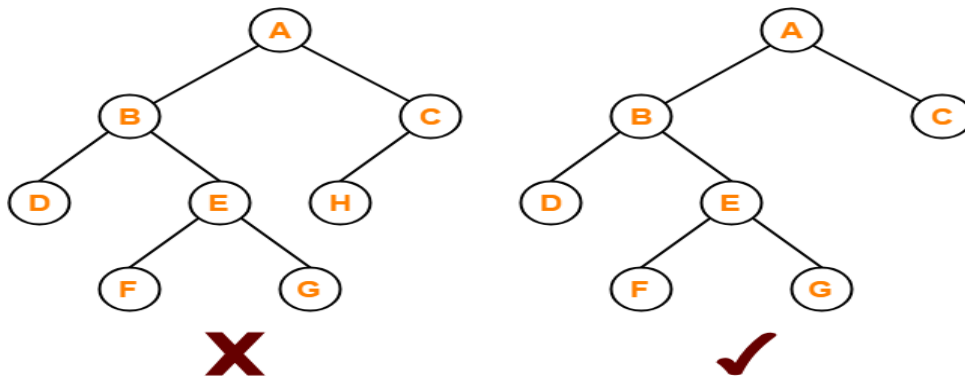
Complete Binary Tree: A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

Example-
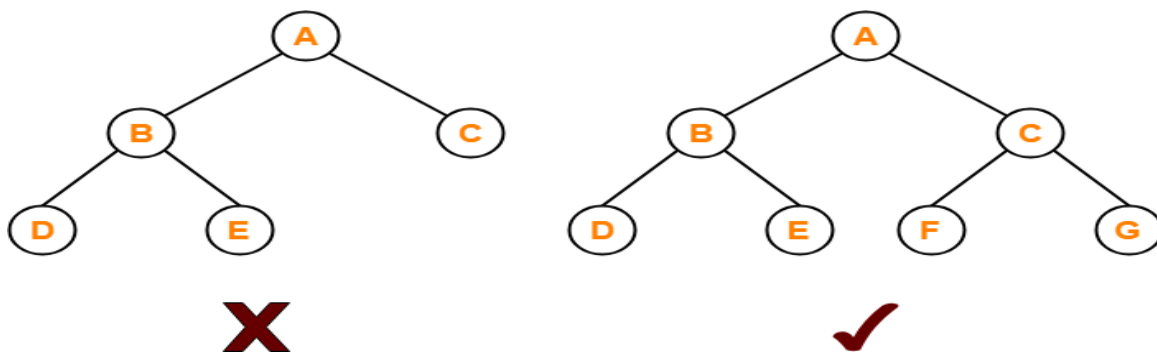
Binary tree representation and traversal

Trees can be represented in two ways as listed below:

1. Dynamic Node Representation (Linked Representation).

2. Array Representation (Sequential Representation).

Dynamic Node Representation (Linked Representation).

Binary trees in linked representation are stored in the memory as linked lists. These lists have nodes that aren't stored at adjacent or neighbouring memory locations and are linked to each other through the parent-child relationship associated with trees.

In this representation, each node has three different parts –

- pointer that points towards the right node,
- pointer that points towards the left node,
- data element.

Array Representation (Sequential Representation).

This is the more common representation. All binary trees consist of a root pointer that points in the direction of the root node. When you see a root node pointing towards null or 0, you should know that you are dealing with an empty binary tree. The right and left pointers store the address of the right and left children of the tree.

Now, we are going to talk about the sequential representation of the trees. In order to represent a tree using an array, the numbering of nodes can start either from 0–(n-1) or 1–n, consider the below illustration as follows:

Illustration:

```
A(0)
  / \
B(1) C(2)
 / \   \
D(3) E(4)  F(6)
A(1)
   /      \
B(2)      C(3)
 / \   \
D(4) E(5)      F(7)
```
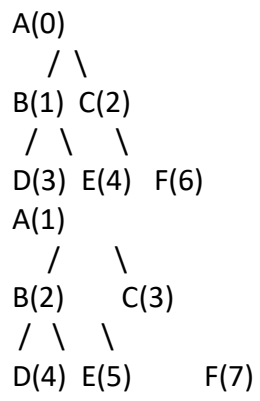
Note: *father, left_child and right_child are the values of indices of the array.*
 Case 1: (0—n-1)
if (say)father=p;
then left_child=(2*p)+1;
and right_child=(2*p)+2;
Case 2: 1—n
if (say)father=p;
then left_son=(2*p);
and right_son=(2*p)+1;

Tree traversal :
 tree traversal means **traversing** or **visiting** each node of a tree. Linear data structures like Stack, Queue, linked list have only one way for traversing, whereas the tree has various ways to traverse or visit each node. The following are the three different ways of traversal:
- **Inorder traversal**
- **Preorder traversal**
- **Postorder traversal**
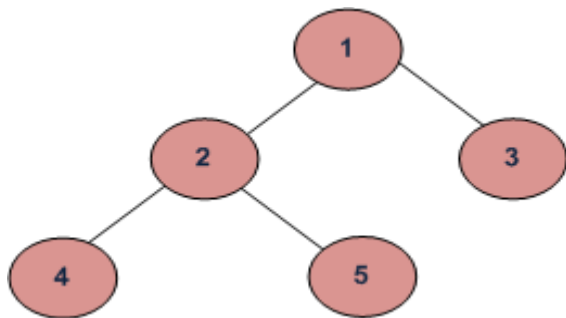
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)

   2. Visit the root.

   3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Ex. inorder traversal for the above-given figure: 4 2 5 1 3
Preorder Traversal (Practice):
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
 Example: Preorder traversal for the above-given figure is 1 2 4 5 3.
PostorderTraversal :
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
Example: Postorder traversal for the above-given figure is 4 5 2 3 1.

BINARY SEARCH TREE
Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.

- The right subtree of a node contains only nodes with keys greater than the node's key.

- The left and right subtree each must also be a binary search tree.

left_subtree (keys) < node (key) ≤ right_subtree (keys)
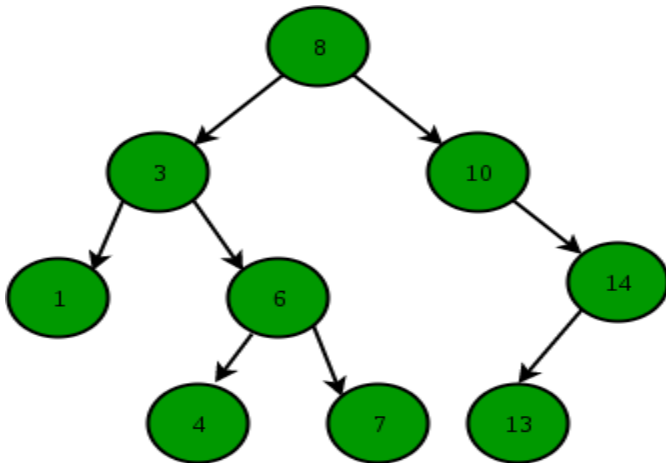
Advantages of Binary search tree

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.
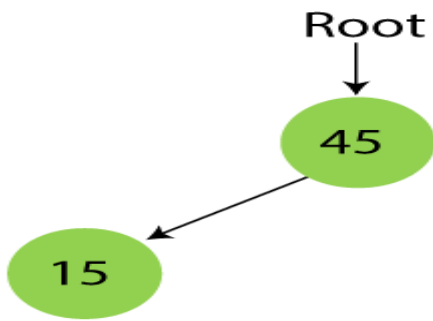
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.
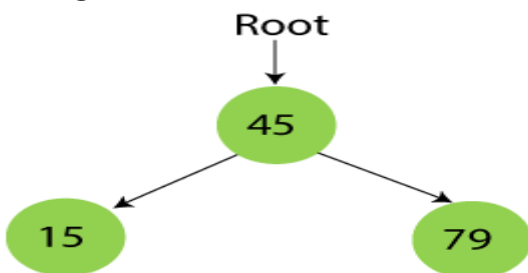
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.
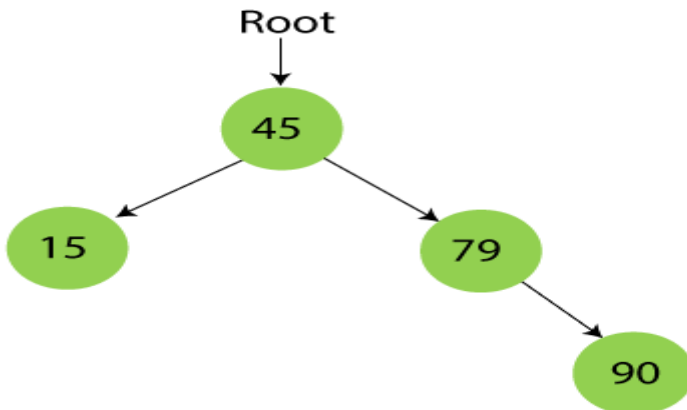
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.
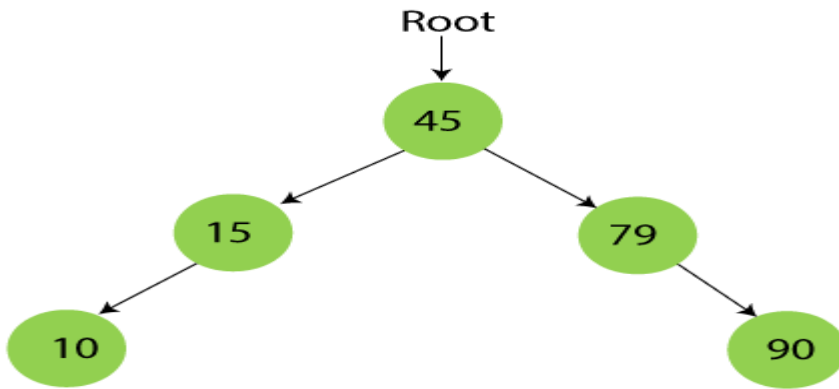
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.
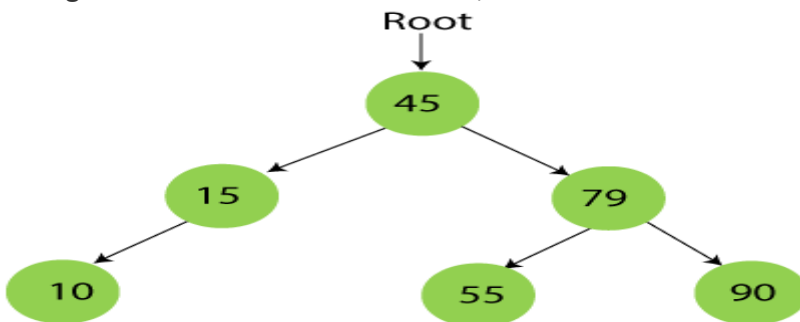
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.
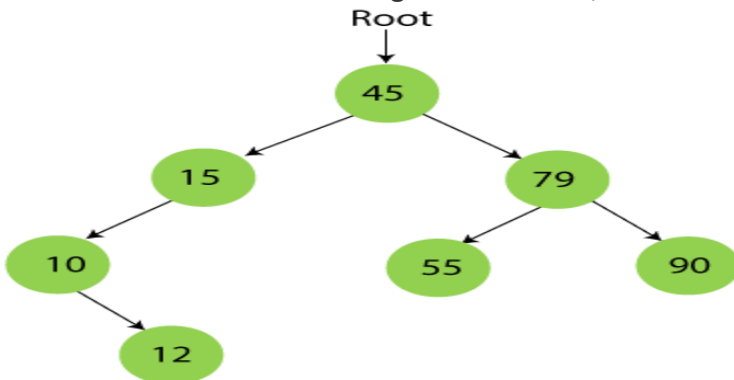
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.
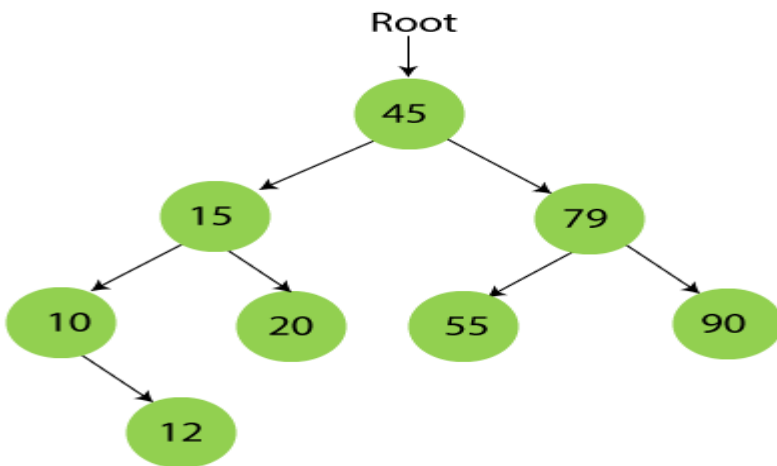
Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.
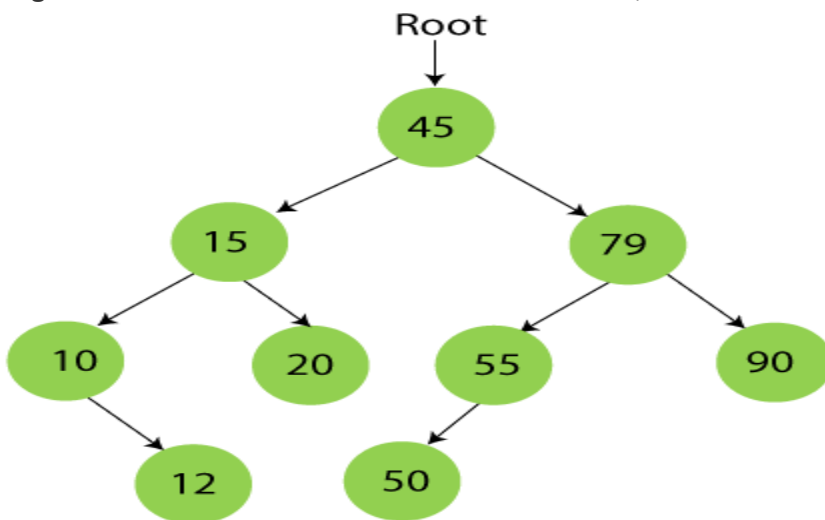
Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

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:
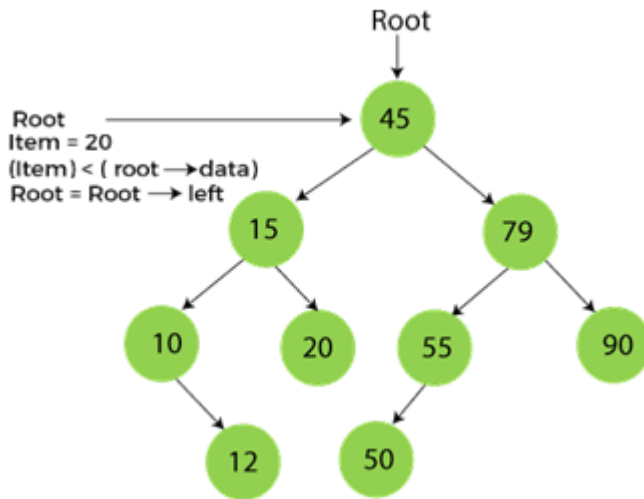
Step2:

Step3:
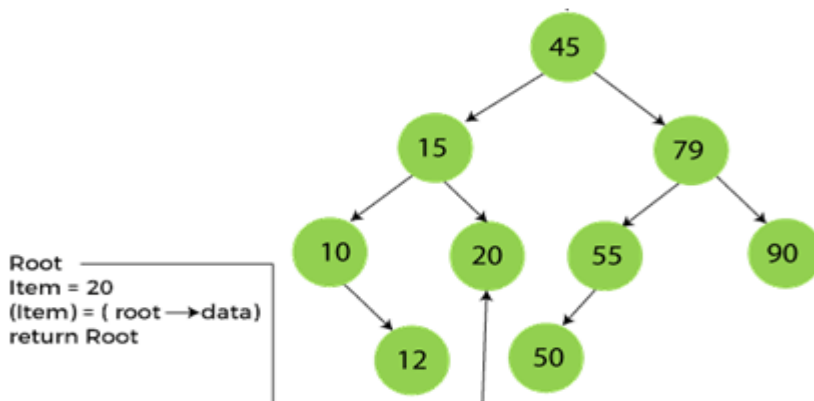
Algorithm to search an element in Binary search tree

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

6.3 Explain insertion & deletion in a binary search trees
Insert Operation
Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.
The insertion of a new key always takes place as the child of some leaf node.
For finding out the suitable leaf node,

- Search the key to be inserted from the root node till some leaf node is reached.

- Once a leaf node is reached, insert the key as child of that leaf node.

Example-

Consider the following example where key = 40 is inserted in the given BST-
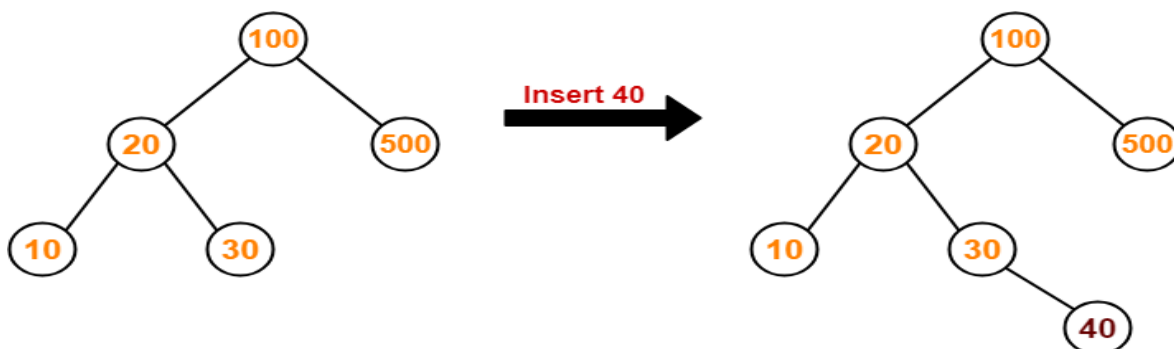
- We start searching for value 40 from the root node 100.

- As 40 < 100, so we search in 100's left subtree.

- As 40 > 20, so we search in 20's right subtree.

- As 40 > 30, so we add 40 to 30's right subtree.

Deletion Operation-

Deletion Operation is performed to delete a particular element from the Binary Search Tree.

When it comes to deleting a node from the binary search tree, following three cases are possible-

Case-01: Deletion Of A Node Having No Child (Leaf Node)-

Just remove / disconnect the leaf node that is to deleted from the tree.

Example-

Consider the following example where node with value = 20 is deleted from the BST-
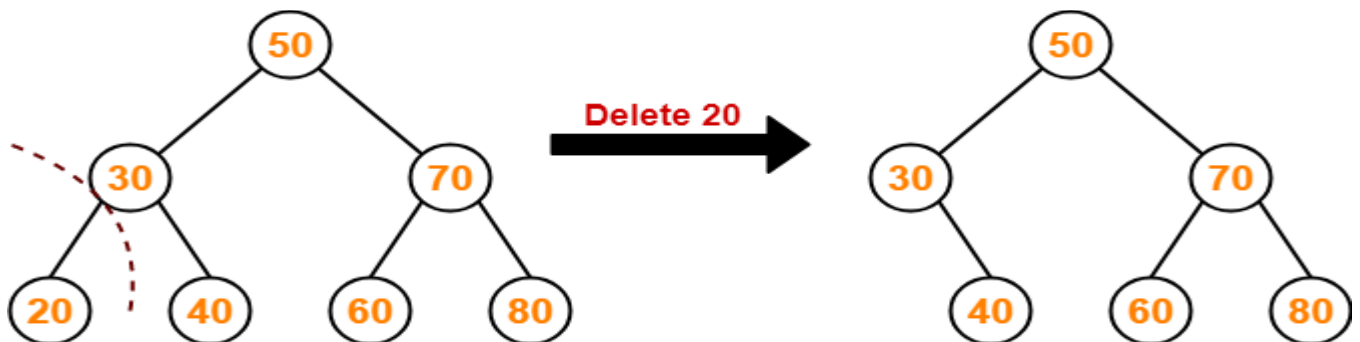
Case-02: Deletion Of A Node Having Only One Child-

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.
Example-

Consider the following example where node with value = 30 is deleted from the BST-
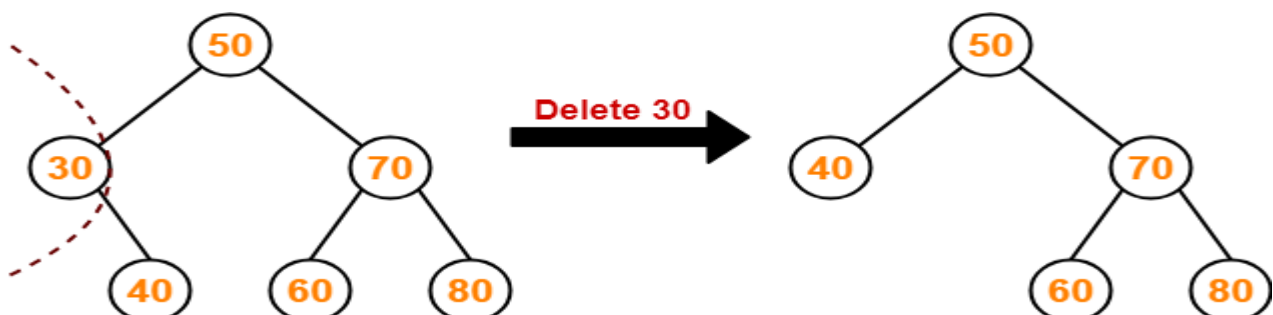
Case-02: Deletion Of A Node Having Two Children-
 It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its **in-order successor or predecessor** recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated

space.
A node with two children may be deleted from the BST in the following two ways-

Method-01:

- Visit to the right subtree of the deleting node.

- Pluck the least value element called as inorder successor.

- Replace the deleting element with its inorder successor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-
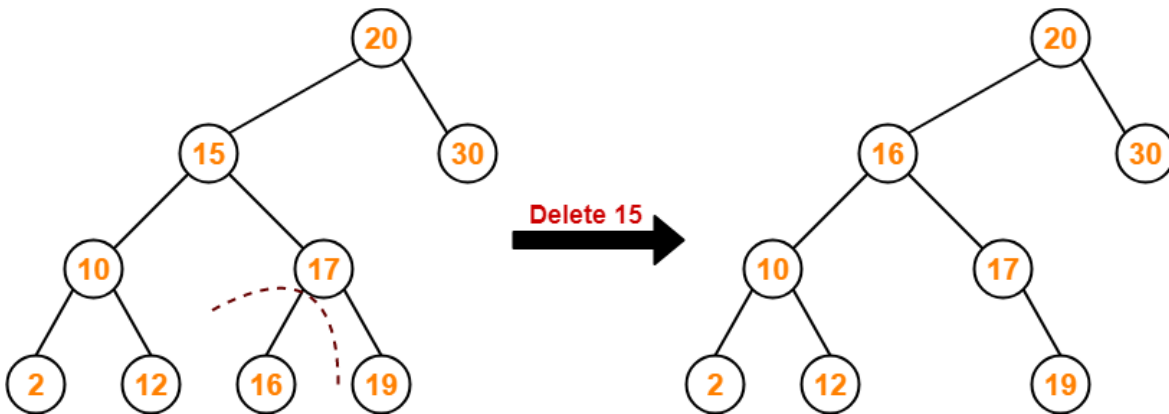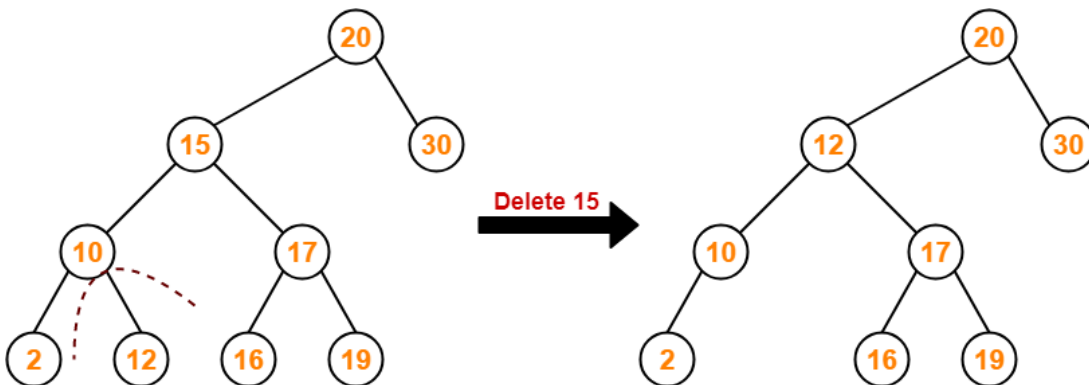

Method-02:
Visit to the left subtree of the deleting node.
- Pluck the greatest value element called as inorder predecessor.

- Replace the deleting element with its inorder predecessor.

Example-
Consider the following example where node with value = 15 is deleted from the BST-

Assignment:

1. Define a tree.
2. Define the terms root, child, subtree , depth , height, level of a tree.
3. Define binary tree.
4. Explain the memory representation of a binary tree.
5. Explain inorder traversal of a binary tree giving an example.
6. Explain preorder traversal of a binary tree giving an example.
7. Explain postorder traversal of a binary tree giving an example.
8. Explain the significance of binary search tree.
9. How can you search an element in a binary search tree. Expain with ex.
10. Use the BST insertion procedure cerate the following elements.

43,56,2,34,55,66,99,77,35
11.in the above BST give the steps to delete 55.
12.How to search for a key in a binary search tree?
13.Here is a small binary tree:

```
    14
   / \
   2   11
  /\  /\
1 3 10 30
 / /
7 40
```
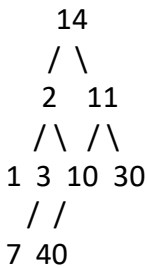
Circle all the leaves. Put a square box around the root. Draw a star around each ancestor of the node
that contains 10. Put a big X through every descendant of the node the contains 1
14.Here is a small binary tree:

```
    14
   / \
   2   11
  /\  /\
1 3 10 30
/ /
7 40
```
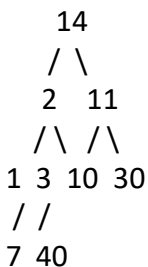
Write the order of the nodes visited in:
A. An in-order traversal:
B. A pre-order traversal:
C. A post-order traversal:
15. in the above tree find the following:
    1. leaves, siblings,
    2. What is the value stored in the parent node of the node containing 30?
    3. How many of the nodes have at least one sibling?
    4. How many descendants does the root have?
    5. What is the depth of the tree?
    6. How many children does the root have?
16.What is the minimum number of nodes in a full binary tree with depth 3?
17.What is the minimum number of nodes in a complete binary tree with depth 3

Binary search tree insertion
Algorithm Insert (TREE, ITEM)
Step 1: IF TREE = NULL
               Allocate memory for TREE
               SET TREE -> DATA = ITEM
               SET TREE -> LEFT = TREE -> RIGHT = NULL
       ELSE
          IF ITEM < TREE -> DATA
               Insert(TREE -> LEFT, ITEM)
         ELSE
               Insert(TREE -> RIGHT, ITEM)
         [END OF IF]
      [END OF IF]
Step 2: END

Algorithm : searching an element in a Binary search tree

Search (ROOT, ITEM)

Step 1: IF ROOT -> DATA = ITEM OR ROOT = NULL
  Return ROOT
  ELSE
  IF ROOT < ROOT -> DATA
  Return search(ROOT -> LEFT, ITEM)
  ELSE
  Return search(ROOT -> RIGHT,ITEM)
  [END OF IF]
  [END OF IF]
Step 2: END

Binary search tree deletion

Algorithm Delete (TREE, ITEM)

Step 1: IF TREE = NULL
          Write "item not found in the tree"
 ELSE IF (ITEM < TREE -> DATA)
       Delete(TREE->LEFT, ITEM)
 ELSE IF(ITEM > TREE -> DATA)
       Delete(TREE -> RIGHT, ITEM)
       ELSE IF (TREE -> LEFT AND TREE -> RIGHT)
               SET TEMP = findLargestNode(TREE -> LEFT)
               SET TREE -> DATA = TEMP -> DATA
               Delete(TREE -> LEFT, TEMP -> DATA)
               ELSE
               SET TEMP = TREE
       IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
               SET TREE = NULL
               ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
                ELSE
               SET TREE = TREE -> RIGHT
 [END OF IF]
  FREE TEMP
[END OF IF]
Step 2: END

7.0 GRAPHS 06

7.1 Explain graph terminology & its representation,

7.2 Explain Adjacency Matrix, Path Matrix

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.
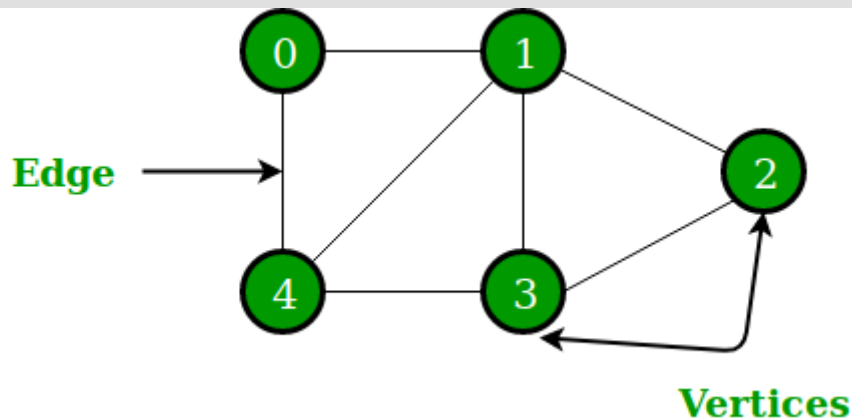
In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Application of graphs

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.

Types of Graph

1. Undirected graph

An undirected graph (graph) is a graph in which edges have no orientation. The edge (x, y) is identical to edge (y, x), i.e., they are not ordered pairs. The maximum number of edges possible in an undirected graph without a loop is n×(n-1)/2.
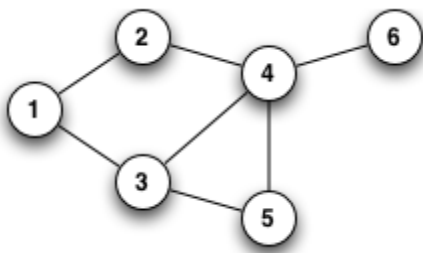
2. Directed graph

A Directed graph (digraph) is a graph in which edges have orientations, i.e., The edge (x, y) is not identical to edge (y, x).
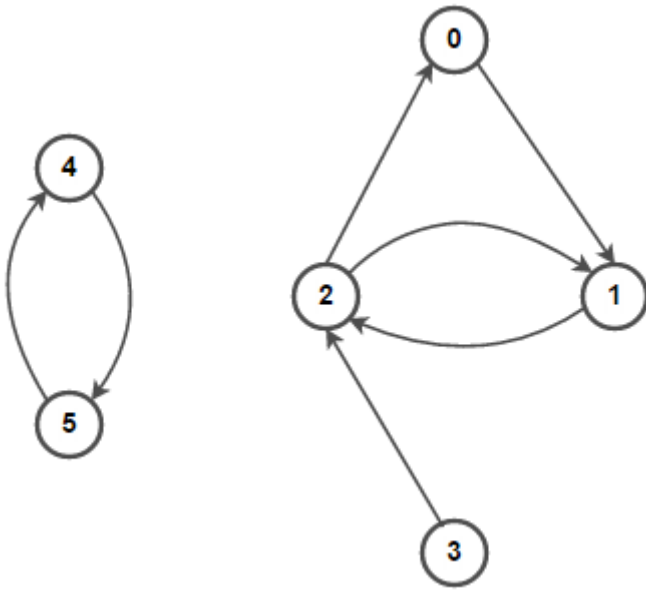
3. Directed Acyclic Graph (DAG)

A Directed Acyclic Graph (DAG) is a directed graph that contains no cycles.
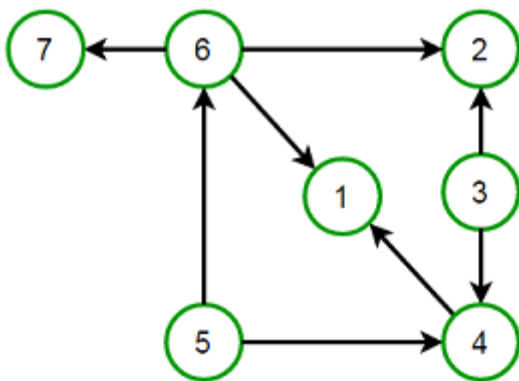

4. Multi graph

A multigraph is an undirected graph in which multiple edges (and sometimes loops) are allowed. Multiple edges are two or more edges that connect the same two vertices. A loop is an edge (directed or undirected) that connects a vertex to itself; it may be permitted or not.

5. Simple graph

A simple graph is an undirected graph in which both multiple edges and loops are disallowed as opposed to a multigraph. In a simple graph with n vertices, every vertex's degree is at most n-1.

Multiple Edges

Loop

Not a Simple Graph

Simple Graph
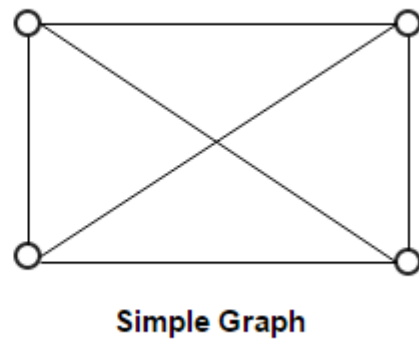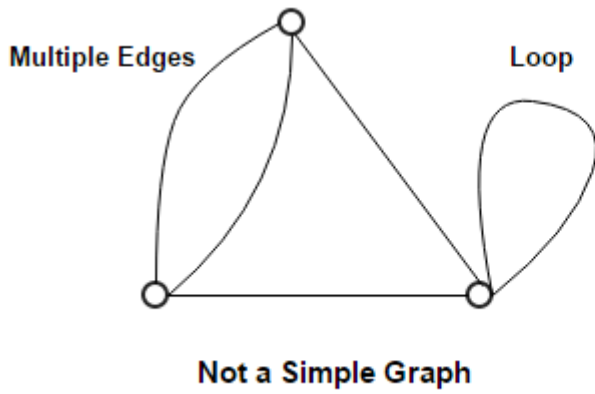
6. Weighted and Unweighted graph

A weighted graph associates a value (weight) with every edge in the graph. We can also use words cost or length instead of weight.

An unweighted graph does not have any value (weight) associated with every edge in the graph. In other words, an unweighted graph is a weighted graph with all edge weight as 1. Unless specified otherwise, all graphs are assumed to be unweighted by default.
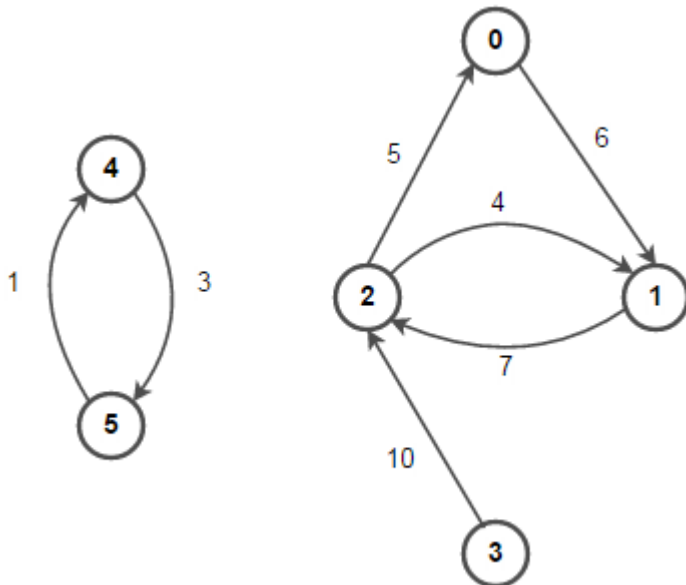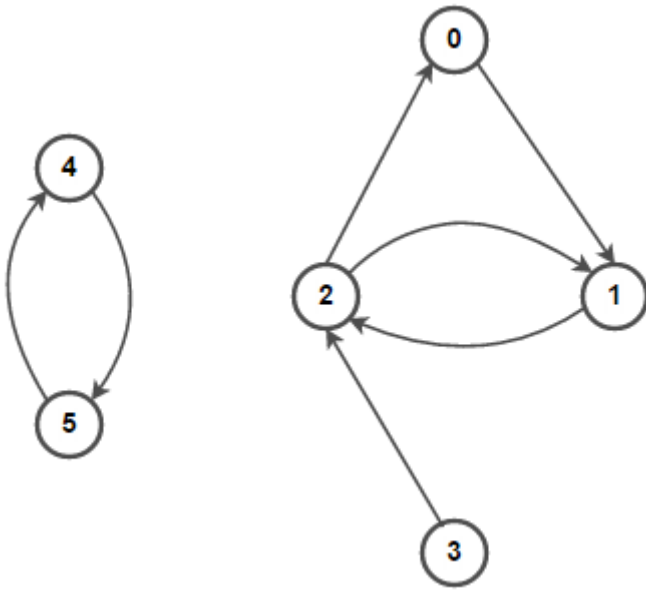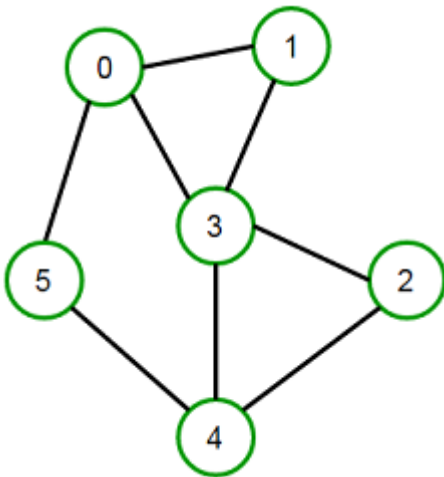
7. Complete graph

A complete graph is one in which every two vertices are adjacent: all edges that could exist are present.
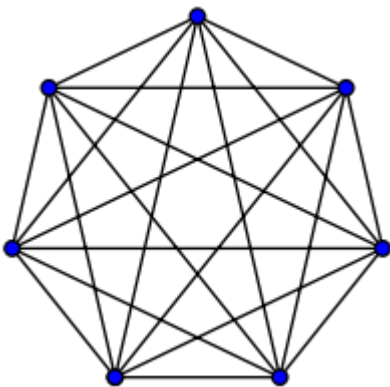
8. Connected graph

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

terms in Graphs

- An edge is (together with vertices) one of the two basic units out of which graphs are constructed. Each edge has two vertices to which it is attached, called its endpoints.
- Two vertices are called adjacent if they are endpoints of the same edge.
- Outgoing edges of a vertex are directed edges that the vertex is the origin.
- Incoming edges of a vertex are directed edges that the vertex is the destination.
- The degree of a vertex in a graph is the total number of edges incident to it.
- In a directed graph, the out-degree of a vertex is the total number of outgoing edges, and the in-degree is the total number of incoming edges.
- A vertex with in-degree zero is called a source vertex, while a vertex with out-degree zero is called a sink vertex.
- An isolated vertex is a vertex with degree zero, which is not an endpoint of an edge.
- Path is a sequence of alternating vertices and edges such that the edge connects each successive vertex.
- Cycle is a path that starts and ends at the same vertex.
- Simple path is a path with distinct vertices.
- A graph is Strongly Connected if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v.
- A directed graph is called Weakly Connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. The vertices in a weakly connected graph have either out-degree or in-degree of at least 1.
- Connected component is the maximal connected subgraph of an unconnected graph.
- A bridge is an edge whose removal would disconnect the graph.
- Forest is a graph without cycles.
- Tree is a connected graph with no cycles. If we remove all the cycles from DAG (Directed Acyclic Graph), it becomes a tree, and if we remove any edge in a tree, it becomes a forest.
- Spanning tree of an undirected graph is a subgraph that is a tree that includes all the vertices of the graph.

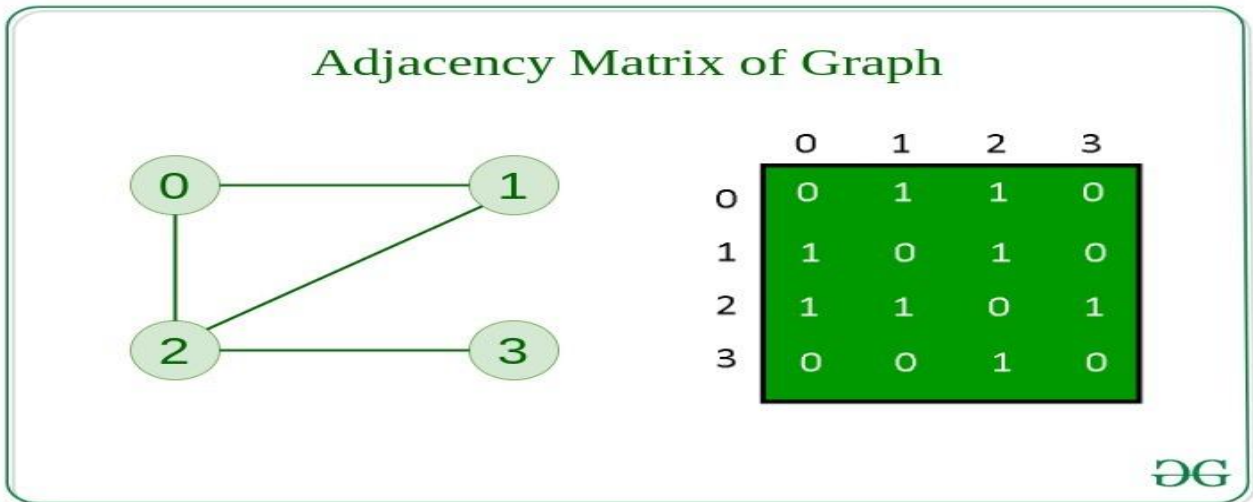### Representation of Graphs

There are two ways to store a graph:

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

In this method, the graph is stored in the form of the 2D matrix where rows and column denote vertices.
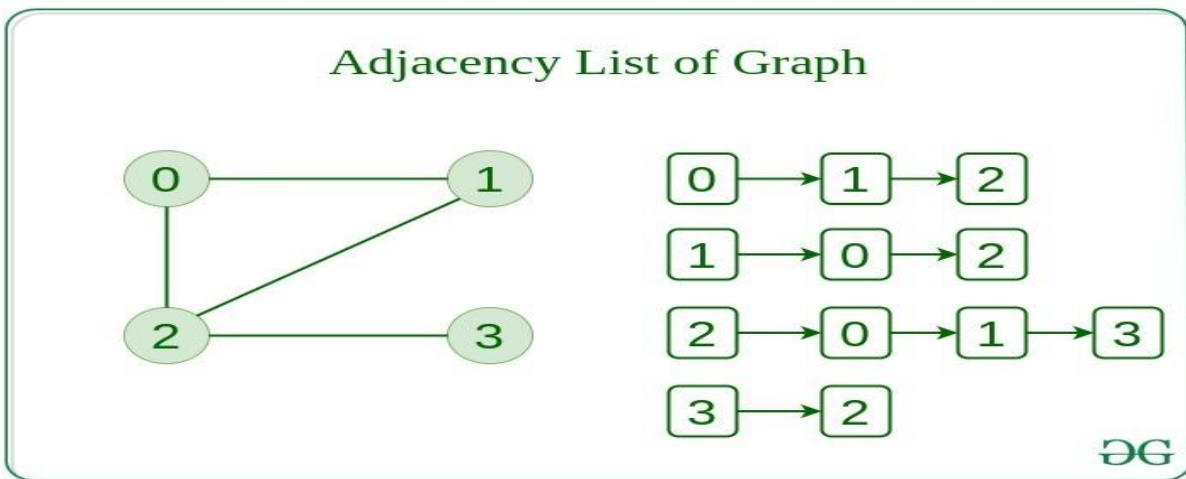Each entry in the matrix represents the weight of the edge between those vertices.

Adjacency Matrix of Graph

Adjacency List

This graph is represented as a collection of linked lists.
There is an array of pointer which points to the edges connected to that vertex.

Adjacency List of Graph

Comparison between them

When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty.
An algorithm such as Prim's and Dijkstra adjacency matrix is used to have less complexity

Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing and edge	O(1)	O(N)

Action	Adjacency Matrix	Adjacency List
Initializing	O(N*N)	O(N)

8.0 SORTING SEARCHING & MERGING
 8.1 Discuss Algorithms for Bubble sort, Quick sort,
8.2 Merging
8.3 Linear searching, Binary searching.

Bubble sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
Example:
First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Assume that A[] is an unsorted array of n elements. This array needs to be sorted in ascending order. The pseudo code is as follows:

```
void bubble_sort( int A[ ], int n ) {
  int temp;
  for(int k = 0; k< n-1; k++) {
    // (n-k-1) is for ignoring comparisons of elements which have already been compared in earlier iterations
    for(int i = 0; i < n-k-1; i++) {
      if(A[ i ] > A[ i+1] ) {
        // here swapping of positions is being done.
        temp = A[ i ];
        A[ i ] = A[ i+1 ];
        A[ i + 1] = temp;
      }
    }
  }
}
```

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 no. of elements.

Quick sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.
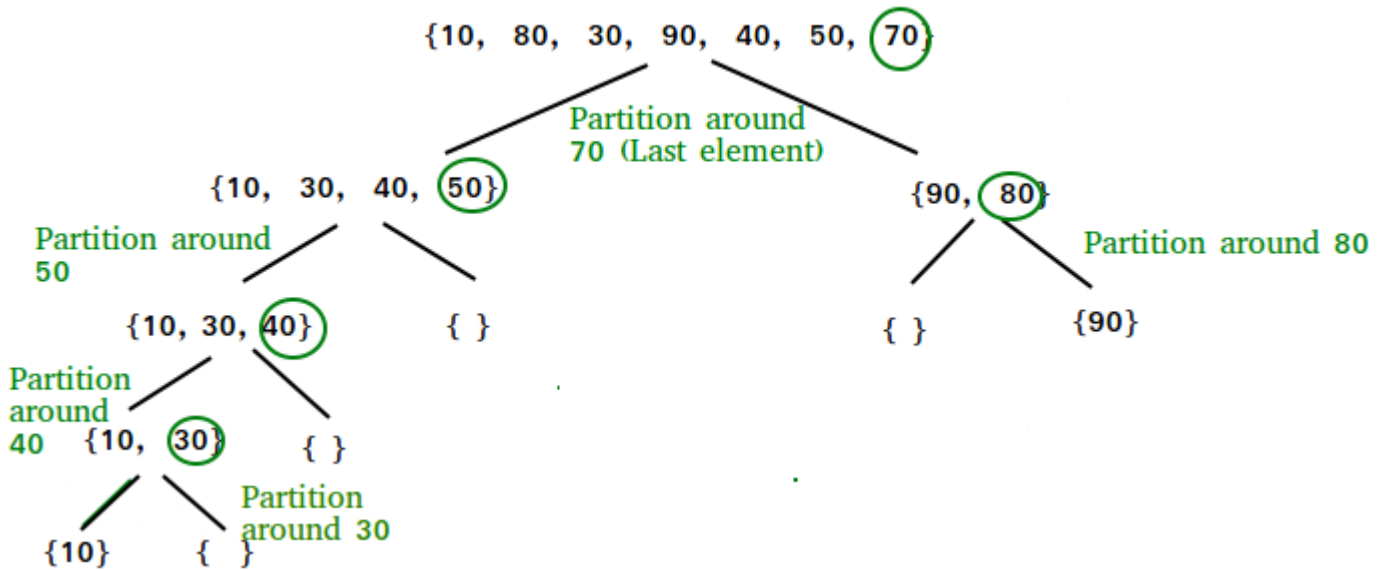
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

{10, 80, 30, 90, 40, 50, (70)}

Partition around
70 (Last element)

{10, 30, 40, (50)}   {90, (80)}

Partition around
50   Partition around 80

{10, 30, (40)}   { }   { }   {90}

Partition
around
40   {10, (30)}   { }

Partition
around 30

{10}   { }

Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

Algorithm:

1. QUICKSORT (array A, start, end)
2. {
3.   1 **if** (start < end)
4.   2 {
5.   3 p = partition(A, start, end)
6.   4 QUICKSORT (A, start, p - 1)
7.   5 QUICKSORT (A, p + 1, end)
8.   6 }
9. }

Partition Algorithm:

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

1. PARTITION (array A, start, end)
2. {
3.   1 pivot ? A[end]
4.   2 i ? start-1
5.   3 **for** j ? start to end -1 {
6.   4 **do if** (A[j] < pivot) {
7.   5 then i ? i + 1
8.   6 swap A[i] with A[j]
9.   7 }}
10. 8 swap A[i+1] with A[end]
11. 9 **return** i+1

12. }

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

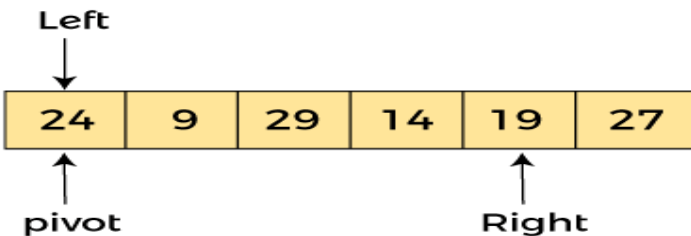Let the elements of array are -

24	9	29	14	19	27

In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

Left

24	9	29	14	19	27

pivot   Right
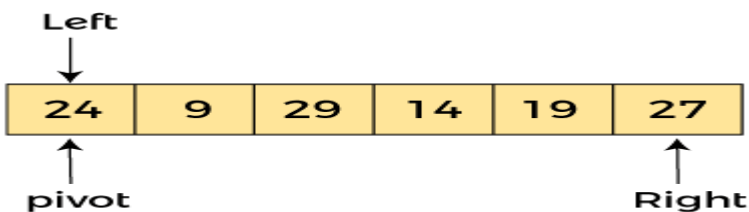
Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -

Left

24	9	29	14	19	27

pivot   Right

Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

pivot
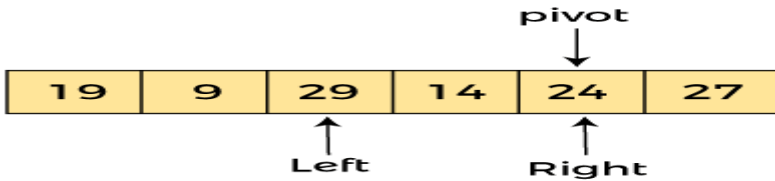
19	9	29	14	24	27
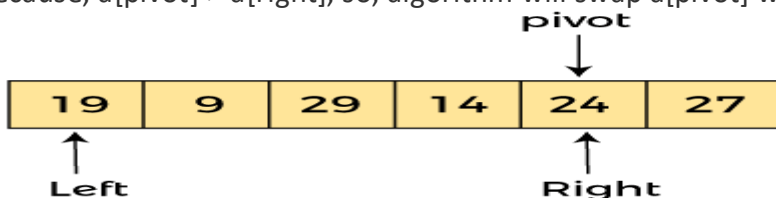
Left   Right

Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.
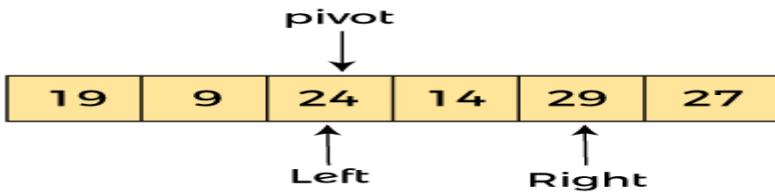
As a[pivot] > a[left], so algorithm moves one position to right as -
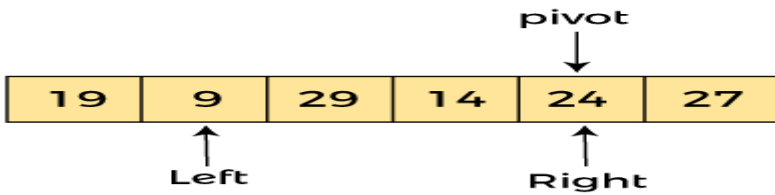
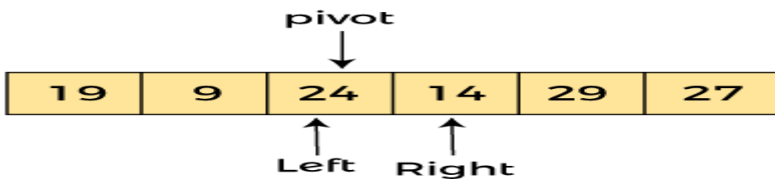Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -
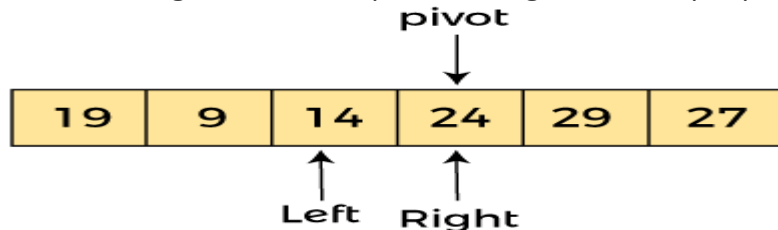

Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -

Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -

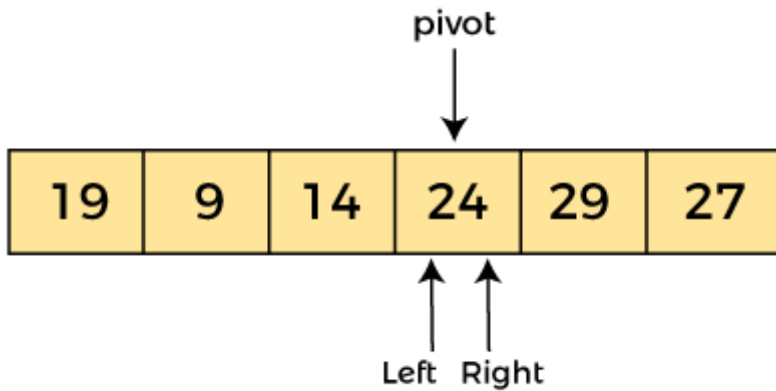Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right],

now pivot is at right, i.e. -

Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.

pivot

| 19 | 9 | 14 | 24 | 29 | 27 |

Left   Right

Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.

| 19 | 9 | 14 | 24 | 29 | 27 |

Left sub array   Right sub array
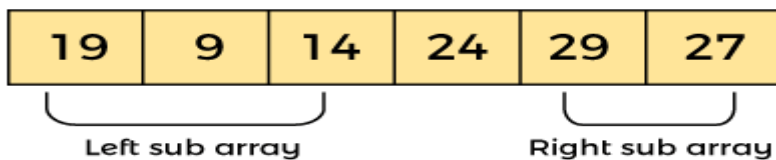
Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -

| 9 | 14 | 19 | 24 | 27 | 29 |

Quicksort complexity

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

1. Time Complexity

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

- o **Best Case Complexity** - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.

- **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 quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **O(n²)**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

2. Space Complexity

Space Complexity	O(n*logn)
Stable	NO

The space complexity of quicksort is O(n*logn).

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.

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

1. MERGE_SORT(arr, beg, end)
2.
3. **if** beg < end
4. set mid = (beg + end)/2
5. MERGE_SORT(arr, beg, mid)
6. MERGE_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of **if**
9.
10. END MERGE_SORT

The following diagram from shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.

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

Let the elements of array are -

| 12 | 31 | 25 | 8 | 32 | 17 | 40 | 42 |
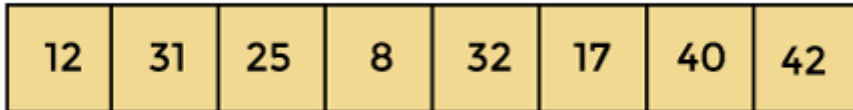
According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

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

divide

| 12 | 31 | 25 | 8 | | 32 | 17 | 40 | 42 |

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

divide

| 12 | 31 | | 25 | 8 | | 32 | 17 | | 40 | 42 |
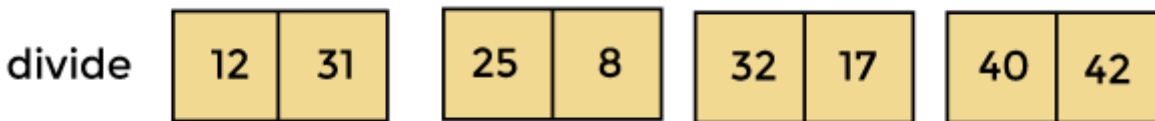
Now, again divide these arrays to get the atomic value that cannot be further divided.

divide

| 12 | | 31 | | 25 | | 8 | | 32 | | 17 | | 40 | | 42 |

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

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

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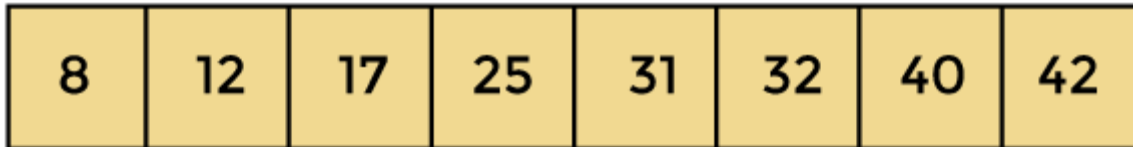
merge

| 12 | 31 | | 8 | 25 | | 17 | 32 | | 40 | 42 |
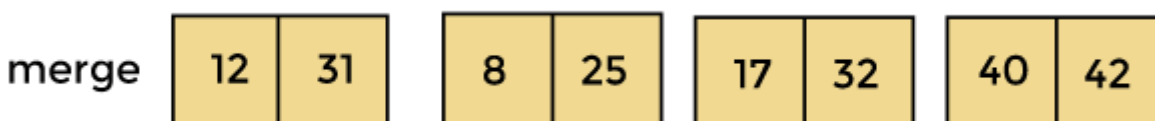
In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.

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

Now, the array is completely sorted.

Merge sort complexity

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

1. Time Complexity

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

- o **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- o **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 merge sort is **O(n*logn)**.
- o **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 merge sort is **O(n*logn)**.

2. Space Complexity

Space Complexity	O(n)
Stable	YES

- o The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

3 Linear searching, Binary searching

Linear searching

Linear 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.

procedure linear_search (list, value)

```
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for

end procedure
```

Input : arr[] = {10, 20, 80, 30, 60, 50,
            110, 100, 130, 170}
      x = 110;
Output : 6
Element x is present at index 6

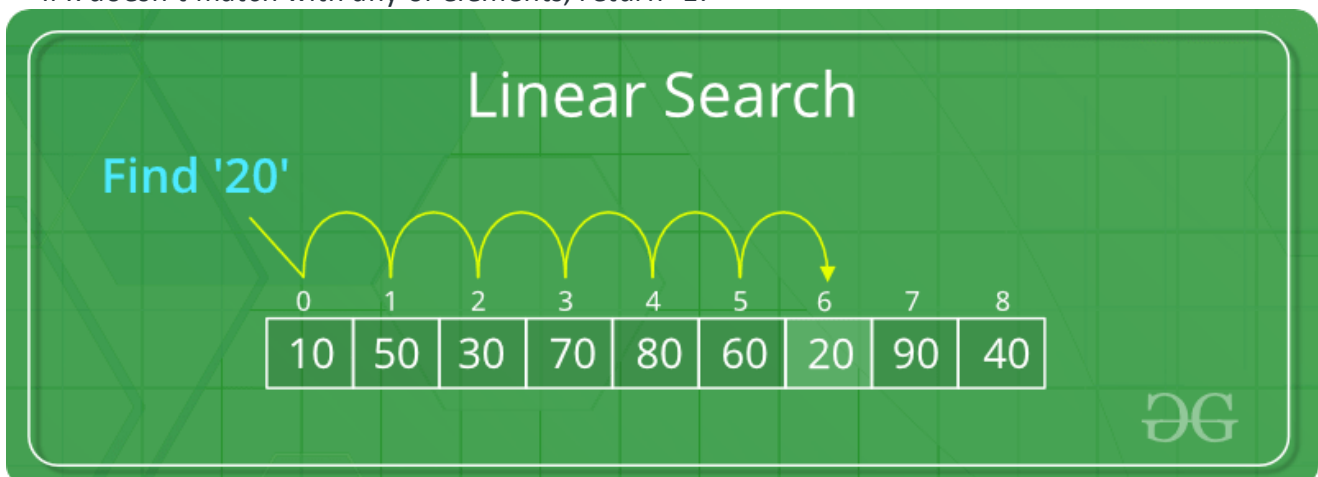**Input :** arr[] = {10, 20, 80, 30, 60, 50,
            110, 100, 130, 170}
      x = 175;
Output : -1
Element x is not present in arr[].
A simple approach is to do a **linear search**, i.e
- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

The **time complexity** of the above algorithm is O(n).
Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster-searching comparison to Linear search.

Improve Linear Search Worst-Case Complexity
1. if element Found at last O(n) to O(1)

2. It is the same as previous method because here we are performing 2 'if' operations in one iteration of the loop and in previous method we performed only 1 'if' operation. This makes both the time complexities same.

Binary search

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 on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

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 –

mid = low + (high - low) / 2
Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.
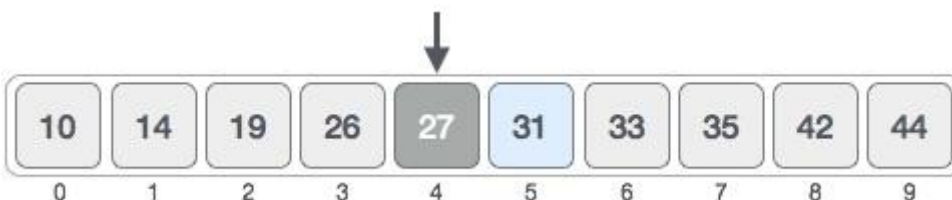
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.
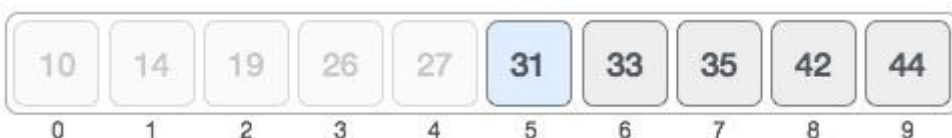
We change our low to mid + 1 and find the new mid value again.

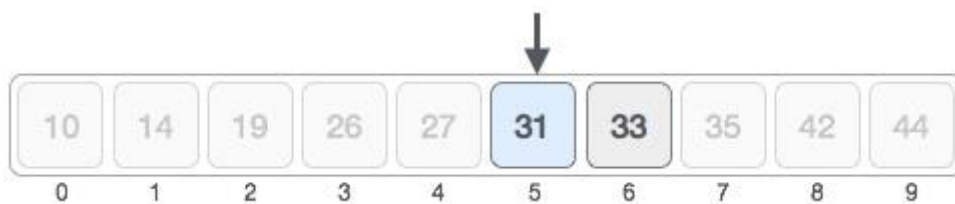low = mid + 1
mid = low + (high - low) / 2
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

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 calculate the mid again. This time it is 5.

We compare the value stored at location 5 with our target value. We find that it is a match.

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

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

Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + (upperBound - lowerBound) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x

```
      set upperBound = midPoint - 1

   if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

programming assignment:

Implementation of merge sort

Now, let's see the programs of merge sort in different programming languages.

Program: Write a program to implement merge sort in C language.

```c
1.  #include <stdio.h>
2.
3.  /* Function to merge the subarrays of a[] */
4.  void merge(int a[], int beg, int mid, int end)
5.  {
6.      int i, j, k;
7.      int n1 = mid - beg + 1;
8.      int n2 = end - mid;
9.
10.     int LeftArray[n1], RightArray[n2]; //temporary arrays
11.
12.     /* copy data to temp arrays */
13.     for (int i = 0; i < n1; i++)
14.     LeftArray[i] = a[beg + i];
15.     for (int j = 0; j < n2; j++)
16.     RightArray[j] = a[mid + 1 + j];
17.
18.     i = 0; /* initial index of first sub-array */
19.     j = 0; /* initial index of second sub-array */
20.     k = beg;  /* initial index of merged sub-array */
21.
22.     while (i < n1 && j < n2)
23.     {
24.         if(LeftArray[i] <= RightArray[j])
25.         {
26.             a[k] = LeftArray[i];
27.             i++;
28.         }
29.         else
30.         {
31.             a[k] = RightArray[j];
32.             j++;
33.         }
34.         k++;
```

```c
35.    }
36.    while (i<n1)
37.    {
38.        a[k] = LeftArray[i];
39.        i++;
40.        k++;
41.    }
42.
43.    while (j<n2)
44.    {
45.        a[k] = RightArray[j];
46.        j++;
47.        k++;
48.    }
49. }
50.
51. void mergeSort(int a[], int beg, int end)
52. {
53.    if (beg < end)
54.    {
55.        int mid = (beg + end) / 2;
56.        mergeSort(a, beg, mid);
57.        mergeSort(a, mid + 1, end);
58.        merge(a, beg, mid, end);
59.    }
60. }
61.
62. /* Function to print the array */
63. void printArray(int a[], int n)
64. {
65.    int i;
66.    for (i = 0; i < n; i++)
67.        printf("%d ", a[i]);
68.    printf("\n");
69. }
70.
71. int main()
72. {
73.    int a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
74.    int n = sizeof(a) / sizeof(a[0]);
75.    printf("Before sorting array elements are - \n");
76.    printArray(a, n);
77.    mergeSort(a, 0, n - 1);
78.    printf("After sorting array elements are - \n");
79.    printArray(a, n);
80.    return 0;
81. }
```

Output:

```
Before sorting array elements are -
12 31 25 8 32 17 40 42
After sorting array elements are -
8 12 17 25 31 32 40 42
```

Implementation of quicksort

Now, let's see the programs of quicksort in different programming languages.

Program: Write a program to implement quicksort in C language.

```c
1.  #include <stdio.h>
2.  /* function that consider last element as pivot,
3.  place the pivot at its exact position, and place
4.  smaller elements to left of pivot and greater
5.  elements to right of pivot.  */
6.  int partition (int a[], int start, int end)
7.  {
8.      int pivot = a[end]; // pivot element
9.      int i = (start - 1);
10.
11.     for (int j = start; j <= end - 1; j++)
12.     {
13.         // If current element is smaller than the pivot
14.         if (a[j] < pivot)
15.         {
16.             i++; // increment index of smaller element
17.             int t = a[i];
18.             a[i] = a[j];
19.             a[j] = t;
20.         }
21.     }
22.     int t = a[i+1];
23.     a[i+1] = a[end];
24.     a[end] = t;
25.     return (i + 1);
26. }
27.
28. /* function to implement quick sort */
29. void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end = Ending index */
30. {
31.     if (start < end)
32.     {
33.         int p = partition(a, start, end); //p is the partitioning index
34.         quick(a, start, p - 1);
```

```
35.        quick(a, p + 1, end);
36.    }
37. }
38.
39. /* function to print an array */
40. void printArr(int a[], int n)
41. {
42.    int i;
43.    for (i = 0; i < n; i++)
44.        printf("%d ", a[i]);
45. }
46. int main()
47. {
48.    int a[] = { 24, 9, 29, 14, 19, 27 };
49.    int n = sizeof(a) / sizeof(a[0]);
50.    printf("Before sorting array elements are - \n");
51.    printArr(a, n);
52.    quick(a, 0, n - 1);
53.    printf("\nAfter sorting array elements are - \n");
54.    printArr(a, n);
55.
56.    return 0;
57. }
```

Output:

```
Before sorting array elements are -
24 9 29 14 19 27
After sorting array elements are -
9 14 19 24 27 29
```

Programming

```cpp
// C++ program for implementation of Bubble sort

#include <bits/stdc++.h>

using namespace std;



void swap(int *xp, int *yp)

{

    int temp = *xp;
```

```cpp
    *xp = *yp;

    *yp = temp;

}


// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

    int i, j;

    for (i = 0; i < n-1; i++)


    // Last i elements are already in place

    for (j = 0; j < n-i-1; j++)

        if (arr[j] > arr[j+1])

            swap(&arr[j], &arr[j+1]);

}


/* Function to print an array */

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << arr[i] << " ";

    cout << endl;
```

```
}


// Driver code

int main()

{

    int arr[] = {64, 34, 25, 12, 22, 11, 90};

    int n = sizeof(arr)/sizeof(arr[0]);

    bubbleSort(arr, n);

    cout<<"Sorted array: \n";

    printArray(arr, n);

    return 0;

}


// This code is contributed by rathbhupendra
```

Output:
Sorted array:
11 12 22 25 34 64 90

- **Linear search**

```
// C++ code to linearly search x in arr[]. If x

// is present then return its location, otherwise

// return -1


#include <iostream>
```

```cpp
using namespace std;

int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    int result = search(arr, n, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
```

```
    return 0;

}
```

Output
Element is present at index 3

// C++ program to implement recursive Binary Search

```cpp
#include <bits/stdc++.h>

using namespace std;



// A recursive binary search function. It returns

// location of x in given array arr[l..r] is present,

// otherwise -1

int  binarySearch(int arr[], int l, int r, int x)

{

   if (r >= l) {

      int mid = l + (r - l) / 2;



      // If the element is present at the middle

      // itself

      if (arr[mid] == x)

         return mid;



      // If element is smaller than mid, then

      // it can only be present in left subarray
```

```cpp
        if (arr[mid] > x)

            return binarySearch(arr, l, mid - 1, x);


        // Else the element can only be present

        // in right subarray

        return binarySearch(arr, mid + 1, r, x);

    }


    // We reach here when element is not

    // present in array

    return -1;

}


int main(void)

{

    int arr[] = { 2, 3, 4, 10, 40 };

    int x = 10;

    int n = sizeof(arr) / sizeof(arr[0]);

    int result = binarySearch(arr, 0, n - 1, x);

    (result == -1)

        ? cout << "Element is not present in array"

        : cout << "Element is present at index " << result;

    return 0;
```

}

Output :

Element is present at index 3

Here you can create a check function for easier implementation.