DNYANSAGAR ARTS AND COMMERCE COLLEGE, BALEWADI, PUNE – 45

Subject : Data Structure (sub code CA-302 CBCS 2019 Pattern)

Class : F.Y. BBA(CA)

Prof . S. B. Potadar

www.dacc.edu.in

Unit 1 Basic Structure and Introduction to Data structure

1.1 Pointers and Dynamic Memory allocation

1.2 Algorithm-Definition and characteristics

1.3 Algorithm Analysis -Space Complexity -Time Complexity -
Asymptotic Notation

Introduction to Data structure

1.4 Types of Data structure

1.5 Abstract Data Types (ADT) Introduction to Arrays and Structure

1.6 Types of array and Representation of array

1.7 Polynomial - Polynomial Representation - Evaluation of Polynomial
- Addition of Polynomial

1.8 Self Referential Structure

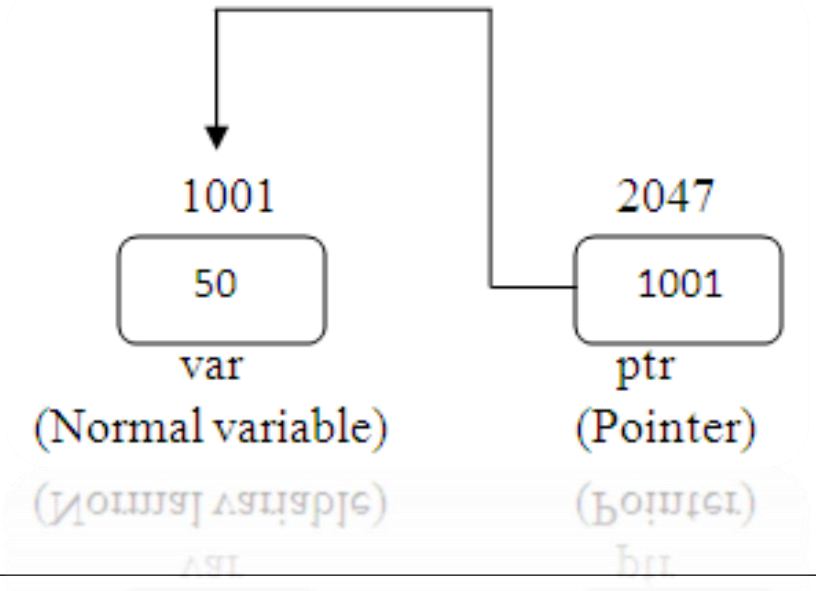
1.1 Pointers and Dynamic Memory allocation

Pointer:

A pointer is used as a referencing mechanism a pointer provides a way to reference an object using that object's address. There are usually three elements involved in this referencing process, a pointer variable, an address, and another variable .The pointer variable holds the address of the other variable. A special operation, called an indirection operation will use this address to actually reference the other variable,

a) A normal variable 'var' has a memory address of 1001 and holds a value 50.

b) A pointer variable has its own address 2047 but stores 1001, which is the address of the variable 'var'.



Working with pointers:

1) Declaration of a Pointer:

The declaration of a pointer variable takes the following form:

data type *pt_name; This tells the compiler three things about the variable pt_name:

- The asterisk (*) tells that the variable pt_name is a pointer variable.
- pt_name needs a memory location.
- pt_name points to a variable of type data type.

2) Initialization of Pointer variables:

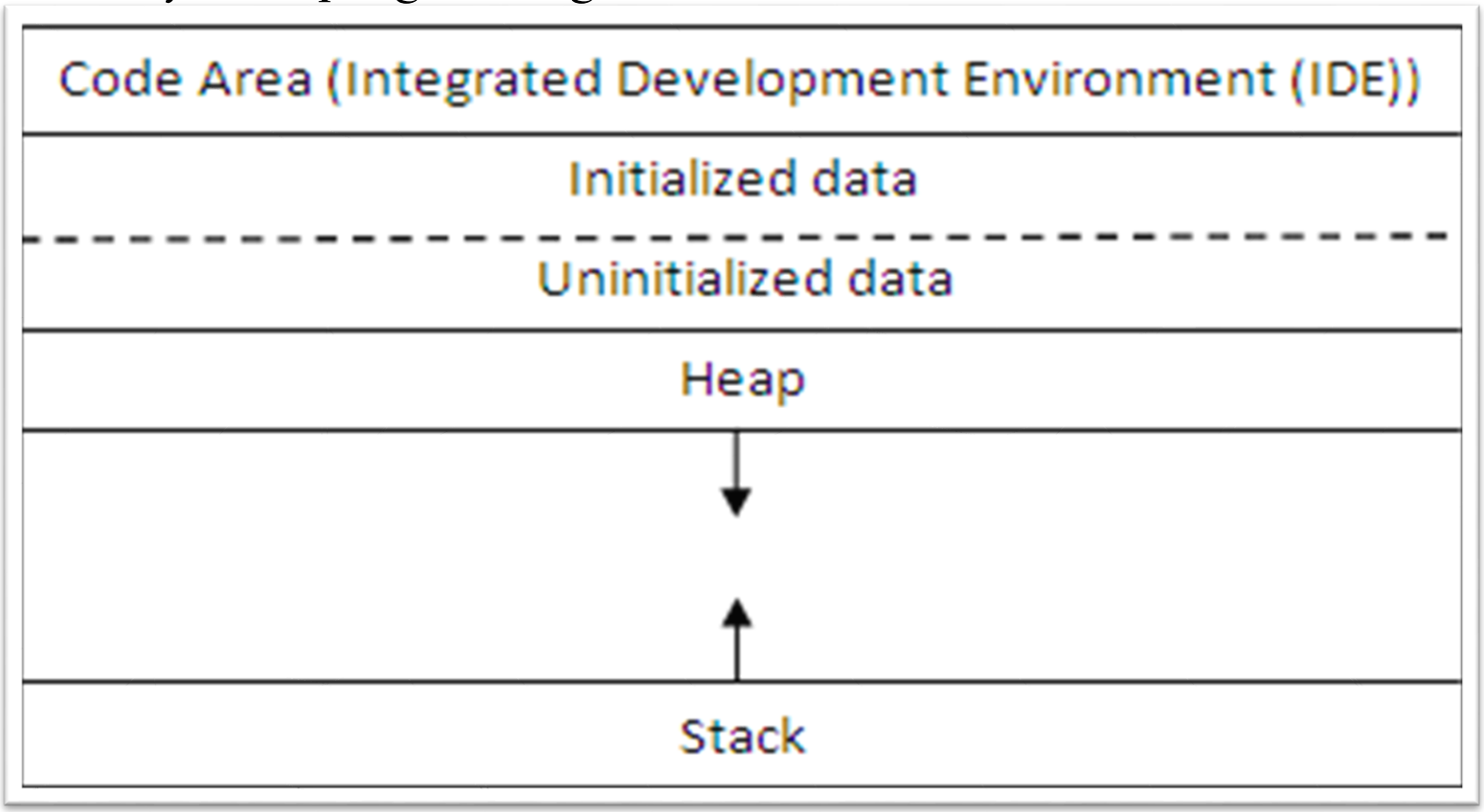
The initialization of the pointer variable is simple like other variable but in the pointer variable the address is assigned to the pointer variable instead of value.

3) Accessing a Variable through its Pointer:

Once a pointer has been assigned the address of a variable, one can access the value of the variable using unary operator '*'(asterisk), known as the indirection operator or dereferencing operator.

C) Dynamic Memory Allocation:

The technique through which a program can be obtaining space in the RAM during the execution of the program and not during compilation is called dynamic memory allocation. The entire runtime view of memory for a program is given below:



Functions of Dynamic Memory Allocation:

1) malloc() function:

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of given size and returns a pointer of typo void. This means that one can assign it to any type of pointer using typecasting. If it fails to locate enough space it returns a NULL pointer.

2) calloc() function:

calloc() is another memory allocation function that is used for allocating memory at runtime. calloc() initializes the allocated memory to zero but, malloc() doesn't. calloc() function is normally used for allocating memory to derived data types such as arrays and structures. If it fails to locate enough space it returns a NULL pointer.

3) realloc() function:

realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size. If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4) free() function:

When a program comes out, operating system automatically release all the memory allocated by the program but as a good practice when there is no need of memory anymore then the memory should be released by calling the function free().free() function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

1.2 Algorithm-Definition and characteristics

Definitions:

1) Alonzo Church and Alan Turing:

"An algorithm is defined as the finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time."

2) David Hilbert:

"An algorithm is a set of instructions designed to perform a specific task."

Characteristics of Algorithm:

1) Finiteness:

An algorithm must terminate after a finite number of steps.

2) Definiteness:

The steps of the algorithm must be precisely defined or unambiguously specified.

3) Correctness:

The output must be true for all input values.

4) Generality:

An algorithm must be generic enough to solve all problems of a particular class.

1.3 Algorithm Analysis : Space Complexity & Time Complexity

Analysis of Algorithm:

1) Considerations in Algorithm Analysis:

Analysis of algorithms focuses on computation of space and time complexity. Space can be defined in terms of space required to store the instructions and data whereas the time is the computer time an algorithm might require for its execution which usually depends on the size of the algorithm and input.

a) Space Complexity:

The space complexity of a problem is a related concept that measures the amount of space, or memory required by the algorithm. Space complexity is measured with Big-O notation.

b) Time Complexity:

Time Complexity is defined as the computer time an algorithm might require for its execution, which usually depends on the size of the algorithm and input.

2) Types of Time Complexities:

a) Best Case Time Complexity:

The best case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size ‘n`.

b) Worst Case Time Complexity:

The worst case time complexity of an algorithm is a measure of the maximum time that the algorithm will require for an input of size ‘n'.

c) Average Case Time Complexity:

The time that an algorithm will require to execute a typical input data of size ‘n' is known as average case time complexity

Fundamentals of Data Structure

- A data structure in Computer Science, is a way of storing and organizing data in a computer's memory or even disk storage so that it can be used efficiently.
- A well-designed data structure allows a variety of critical operations to be performed.
- Data structures are implemented by programming language by the data types, references and operations provide by that particular language.
- Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks.

DATA STRUCTURE AND ITS TYPES

Basically, data structures are of two types

- linear data structure and non linear data structure.

1. Linear data structure :

A data structure is said to be linear if the elements form a sequence i.e., while traversing sequentially, we can reach only one element directly from another.

For example : Array, Linked list,Queue etc.

2. Non linear data structure :

Elements in a nonlinear data structure do not form a sequence i.e each item or element may be connected with two or more other items or elements in a non-linear arrangement. **For example :** Trees and Graphs etc.

DATA STRUCTURE OPERATIONS

- We come to know that data structure is used for the storage of data in computer so that data can be used efficiently.
- The data manipulation within the data structures are performed by means of certain operations.

The following four operations play a major role on data structures.

a) Traversing : Accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “visiting” the record.)

b) Searching : Finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.

c) Inserting : Adding a new record to the structure.

d) Deleting : removing a record from the structure.

Sometimes two or more of these operations may be used in a given situation.

For example, if we want to delete a record with a given key value, at first we will have need to search for the location of the record and then delete that record.

The following two operations are also used in some special situations :

i) Sorting : Operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabetically, with character data.

ii) Merging : combining the records in two different sorted files

into a single sorted file.

Abstract Data Type

A) Meaning:

- An Abstract Data Type (ADT) is a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.
- An abstract data type is the specification of logical and mathematical properties of a data type or structure.
- ADT acts as a useful guideline to implement a data type correctly. The specification of an ADT does not imply any implementation consideration.
- The implementation of an ADT involves the translation of the ADT`s specification into syntax of a particular programming language.
- Thus, ADT involves mainly two parts:
  a) Description of the way in which components are related to each other.
  b) Statements of operations that can be performed on that data

Array : Introduction

- The fundamental data types namely int, float, char etc. are very useful but variable of these data type can be stored only one value at a time. So they can handle limited amount of data. In many applications we need to handle large volume of data for that we have to need powerful data types that would facilitate efficient storing, accessing and manipulation of data items.
- C supports derived data type known as Array.
- Definition-
- An array is collection of data items of the same data type
- An array is fixed-size sequenced collection of elements of the same data type.
- An array is also called as Subscripted Variables
-

Features of Array:

- An array is a collection of similar elements.
- The location of array is the location of its first element.
- The first element in array is numbered zero so the last element is less than the size of array.
- The length of array is the number of its elements in array.
- The type of an array is the data type of its element.
- An array is known as subscripted variable.
- Before using array it's type and dimension must be declared.

Single Dimension Array :

A list of items can be given one variable name using only one subscript and such a variable is called a single subscripted or single Dimension Array.

Syntax :

data type arrayname[size];

Example :

int rollno[3];
float marks[5];
char name[30];

1340 1342 1344 ----- Memory Address

Array elements are always stored in contiguous memory locations and since the data type int occupies 2 bytes of memory, each element will be allocated 2 bytes.

Declaration and Initialization Array :

We can initialise elements of array in the same way the ordinary variable.

Syntax :

data type arrayname[size]={list of values};

Example :

int rollno[5]={1,2,3,4,5}; int rollno[ ]={1,2,3,4,5};
float num[5]={2.5,7.2,9.2,6.2,3.3};

Multi Dimension Array :

An array whose elements are speacified by more than one subscript is known as multi dimension array (also called Matrix)

Syntax :

data type arrayname[row size][column size];

Example :

int student[5][2];

	C0	C1
R0	1	67
R1	2	73
R2	3	82
R3	4	90
R4	5	58

char name [4][10];

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
R0	S	W	A	P	N	I	L	\0		
R1	S	A	N	T	O	S	H	\0		
R2	J	A	Y	D	I	P	\0			
R3	S	A	N	D	I	P	\0			

Declaration and Initialisation Array :

int number[3][4]={8,12,25,37,42,52,68,79,81,92,100,102};

char city[5][10]={“Mumbai”,”Punei”,”Satara”,”Kolhapur”,”Sangli”};

String (Character Array) :

In C character string simply treats as array character. The size in a character string represents the maximum number of characters that the string can hold.

Example :

char name[10];

It declares the name as a character array (string) variable that can hold a maximum 10 characters. Each character of the string is treated as an element of the array name and is stored in the memory as follows.

A Character string terminates with an additional null character. Thus the element in name[10] holds the null character ‘\0’. When declaring character arrays, we must allow one extra element space for the null terminator.

Declaration and Initialisation Array

We can initialise elements of array in the same way as the ordinary variable.

Syntax :

data type arrayname[size]={list of values};

Example :

```
char name[5]={'s','w','a','p','n','i','l','\0'};
char name[]="siddhi";
char name[5]={'P'};
char  *colour[]={"Red","Green","Blue","Yellow"};
```

Polynomial Representation:

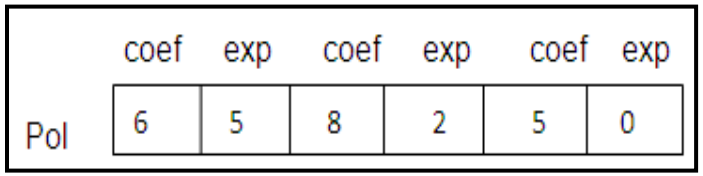
A polynomial of the form $f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_0$ can be considered as a list comprising coefficients and exponents as shown below:

$F(x) = \{ a_n x^n , a_{n-1} x^{n-1} , \ldots . a_1 x^1 , a_{0,} x^0 \}$

For example, the polynomial $6x^5 + 8x^2 + 5$ can be represented as the list shown below:

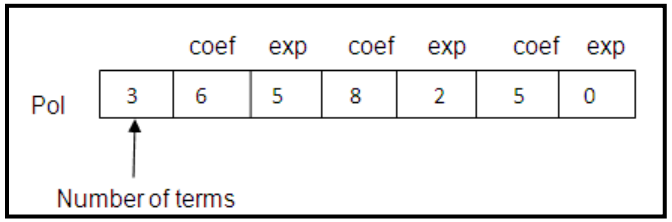
Polynomial = {6, 5, 8, 2, 5, 0}

The above list can be very easily implemented using a one-dimensional array, say Pol [ ] as shown in below figure., where every alternate location contains coefficient and exponent.



Polynomials Representation using Array

The above representation can be modified to incorporate number of terms present in a polynomial by reserving the 0^{th}location of the array for this purpose. The modified representation is shown in below figure. A general polynomial can be represented in an array with descending order of its degree of terms. Therefore, the operations such as addition, multiplication and division on polynomials can be easily carried out.



0^{th}place Reserved for Number of Terms

Evolution of Polynomial:

A generic polynomial is of the form: $p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots\ldots\ldots a_2 x^2 + a_1 x + a_0$. It is assumed that the coefficient values of a_0 through a_n are all known and constant and will be stored in an array. Thus, the evaluation of a polynomial has only the value of x as its input and will return the resulting polynomial value as its output. An alternative method of writing the polynomial is :

$$p(x)\ x = a_0 + a_1 x + a_2 x^2 + \ldots\ldots\ldots + a_{n-1} x^{n-1} + a_n x^n$$

$$= a_0 + x\,(a_1 + x\,(a_2 + x\,(a_3 + x\,(a_4 + \ldots . x(a_{n-1} + x\,a_n)))))$$

The method requires only n multiplication and n additions. The polynomial evaluation can be performed by evaluating the expression in the innermost parenthesis and successfully multiplying by x in for loop. The coefficients $a_0, a_1, a_2, \ldots a_n$ are stored in an array a[n].

Addition of Polynomial:

When adding polynomials only the coefficients of same powerare added and subtracted, the exponents remain unchanged. While adding two polynomials, following cases need to be considered.

1) When the Degrees of Corresponding Terms of the Two Polynomials are Same:

This is the normal case when corresponding coefficients of each term can be added directly.

Example:

$5x^3+2x^2+7$

$7x^3+9x^2+12$

$12x^3+11x^2+19$ is a simple addition where all the degrees of the corresponding terms are same.

2) When the degrees of corresponding terms of the polynomials are different:

The terms with of same power are added but of different powers remain as it is.

$9x^4+5x^3+ 2x$

$3x^4+4x^2+7x$

$12x^4+5x^3+4x^2+9x$

Multiplication of Polynomials:

In general, when multiplying two polynomials together, the distributive property is used, i.e.every term of one polynomial is multiplied with every term of the other polynomial. After that the answer is simplified by combining the like terms. In the following example every term of poly 2 will multiply with every term of poly 1. Coefficients get multiplied and power gets added.

Poly 1:5x^3 + 3x^2+2

Poly 2:3x^2+5x+4

After multiplying, the result is

15x^5 + 9x^4 + 6x^2 + 25x^4 + 15x^3 + 10x + 20x ^3 + 12x^2 + 8

Simplifying the answer by adding the like terms,

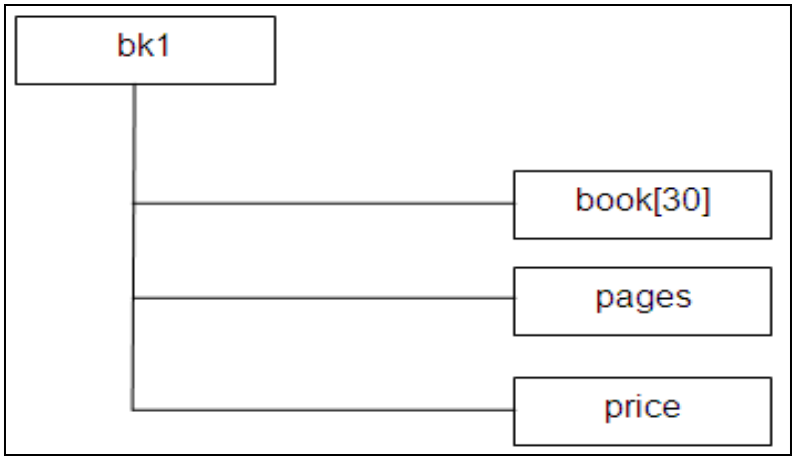
Multiplication result: 15x^5 + 34x^4 + 35x^3 + 18x^2 + 10x + 8

Structure :

Arrays can store many values of a similar data type. Data in the array is of the same composition in nature as far as the type is concerned. To maintain employee's information one should have information such as name, age, qualification, salary and so on. Name and qualification of the employee are char data type, age is an int and salary is float. All these data types cannot be expressed in a single array. One may think to declare different arrays for each data type, but there will be huge increase in source codes of the program. Hence, arrays cannot be useful here. For tackling such a mixed data type problems, a special feature is provided by C known as a structure.

Example: A structure of type book 1 is created. It consists of three members: book [30] of char data type, pages of int type and price of float data type. Figure explains various members of a structure.

```
struct  book1
{
char book[30] ;
int pages;
float price;
} ;
struct book1 bk1;
```



Block Diagram of a Structure

Self-Referential Structure:

b) Declaration of Linked Structure:

The linked structure given in abovefigure can be obtained by the following steps:

Declare structure chain.

Declare variables A and B of type chain.

p(A) = B

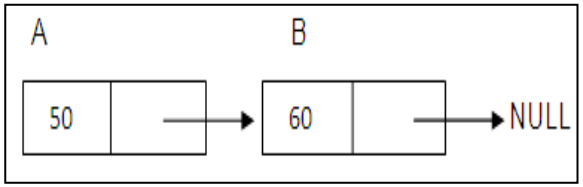
These steps have been coded in the program segment given below:

```
struct chain                    /* declare structure chain */
 {
        int val;
        chain *p:
};
        struct chain A, B;                      /* declare structure variables A and B
        A.p =  &B;              /* Connect  A to B
         B.p = NULL;
```

The data elements in this linked structure can be assigned as follows:

```
        A.val= 50;
        B.val =60;
```

The linked structure now looks like as shown in figure.



Value Assignment to Data Elements

Self-Referential Structure:

c) Advantages of Self-referential Structure:

The self-referential structures have three main advantages over arrays:

1) Flexibility for Memory Allocation:

It is not necessary to know the number of elements and allocate memory in advance for self-referential structures. Memory can be allocated as and when necessary. Insertion and deletion from self-referential structure is efficient using pointers. The individual elements can be scattered anywhere in memory and no contiguous memory is required like array elements.

2) Useful in Programming Constructs:

The self-referential structures are extensively used in programming constructs: linked lists, stacks, Queues and trees etc.

d) typedef keyword:

The C programming language provides a keyword called typedef, which is used to give a type a new name. typedef can be used to give a name to user defined data type. To use typedef with structure define a new data type and then use that data type to define structure variables directly.

Example: typedef unsigned char BYTE;

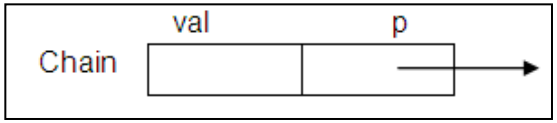
After this type definitions, the identifier BYTE can be used as an abbreviation for the type unsigned char,for example BYTE b1, b2;

Self-Referential Structure:

When a member of a structure is declared as a pointer to the structure itself then the structure is called self-referential structure.

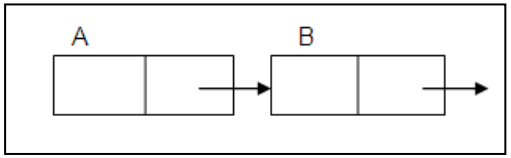
```
struct chain
{
        Intval;
        struct chain *p;
};
```

The structure called 'chain' consists of two members: val and p. The member val is a variable of type int whereas the member p is a pointer to a structure of type chain. Thus, the structure chainhas a member that can point to a structure of type chain or may be itself .This type of self-referencing structure can be viewed as shown in Figure.



Self –referential Structure Chain

Since pointer p can point to a structure variable of type chain, connecting the two such structure variables, A and B, linked structure is obtained as shown in below Figure.



Linked Structure

Unit 2 Linear Data structure

Sorting algorithms with efficiency

-Bubble sort

-Insertion sort

-Merge sort

-Quick Sort

-Selection Sort

2.3 Searching techniques –

Linear Search

Binary search

Searching (Linear and Binary Search)

What is Searching?

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- It can be done on internal data structure or on external data structure.

Searching Techniques

- **To search an element in a given array, it can be done in following ways:**

1. Linear/sequential Search
2. Binary Search

1. Sequential Search

Sequential search is also called as Linear Search.

- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

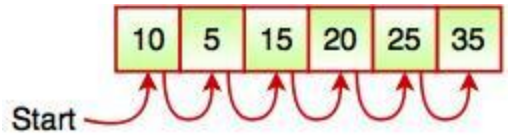


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

2. Binary Search

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.
-

The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

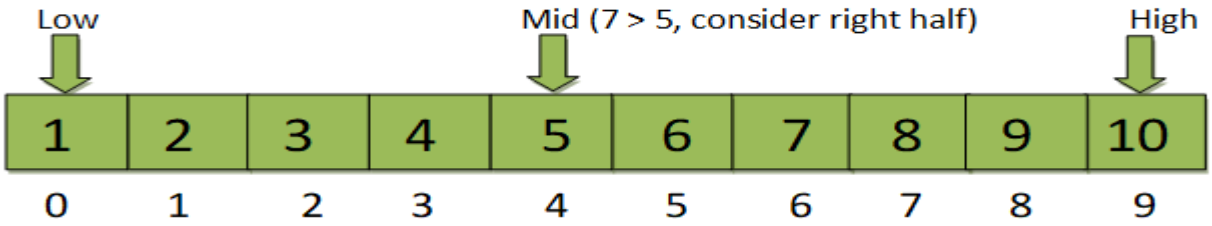
- **For example,** if searching an element 25 in the 7-element array, following figure shows how binary search works:

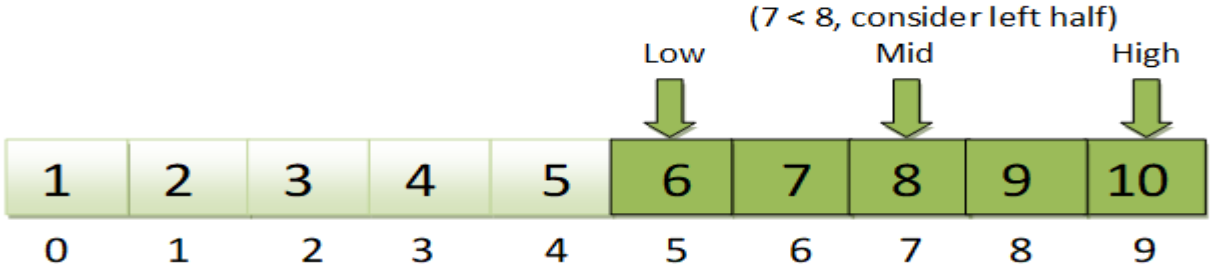
Fig. Working Structure of Binary Search

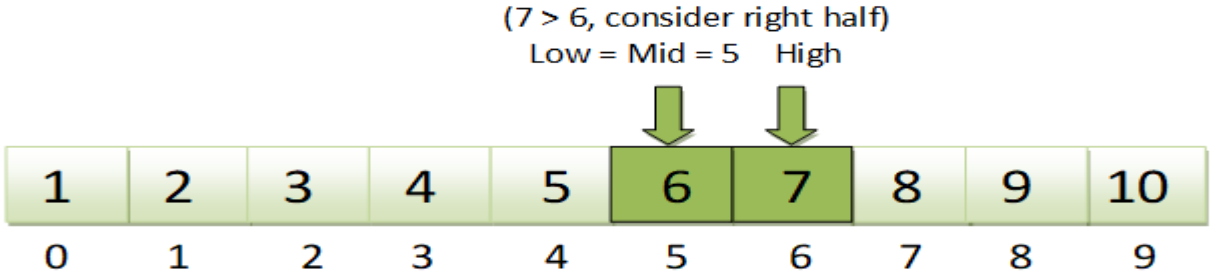
Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the right of the list. Repeat this, till you find an element.

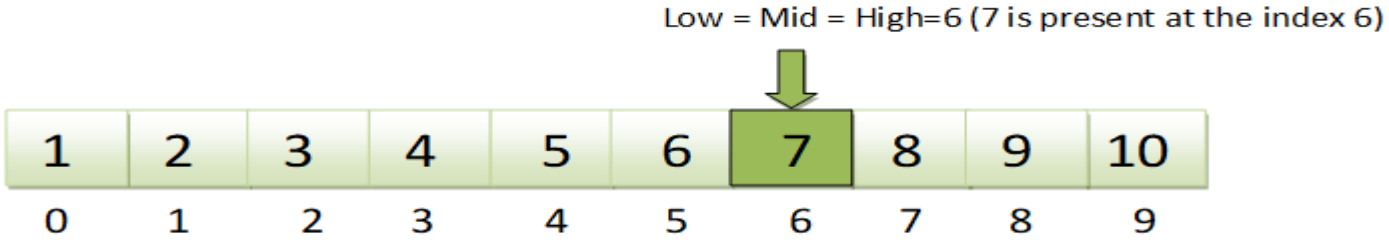
Binary Search

Search the number 7 in the array









What is Sorting?

Sorting is a process of ordering or placing a list of elements from a collection in some kind of order. It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier.

For example, suppose we have a record of employee. It has following data:

Employee No.
Employee Name
Employee Salary
Department Name

Here, employee no. can be takes as key for sorting the records in ascending or descending order. Now, we have to search a Employee with employee no. 116, so we don't require to search the complete record, simply we can search between the Employees with employee no. 100 to 120.Sorting Techniques
Sorting technique depends on the situation. It depends on two parameters.

1. Execution time of program that means time taken for execution of program.
2. Space that means space taken by the program.

Sorting techniques are differentiated by their efficiency and space requirements.

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

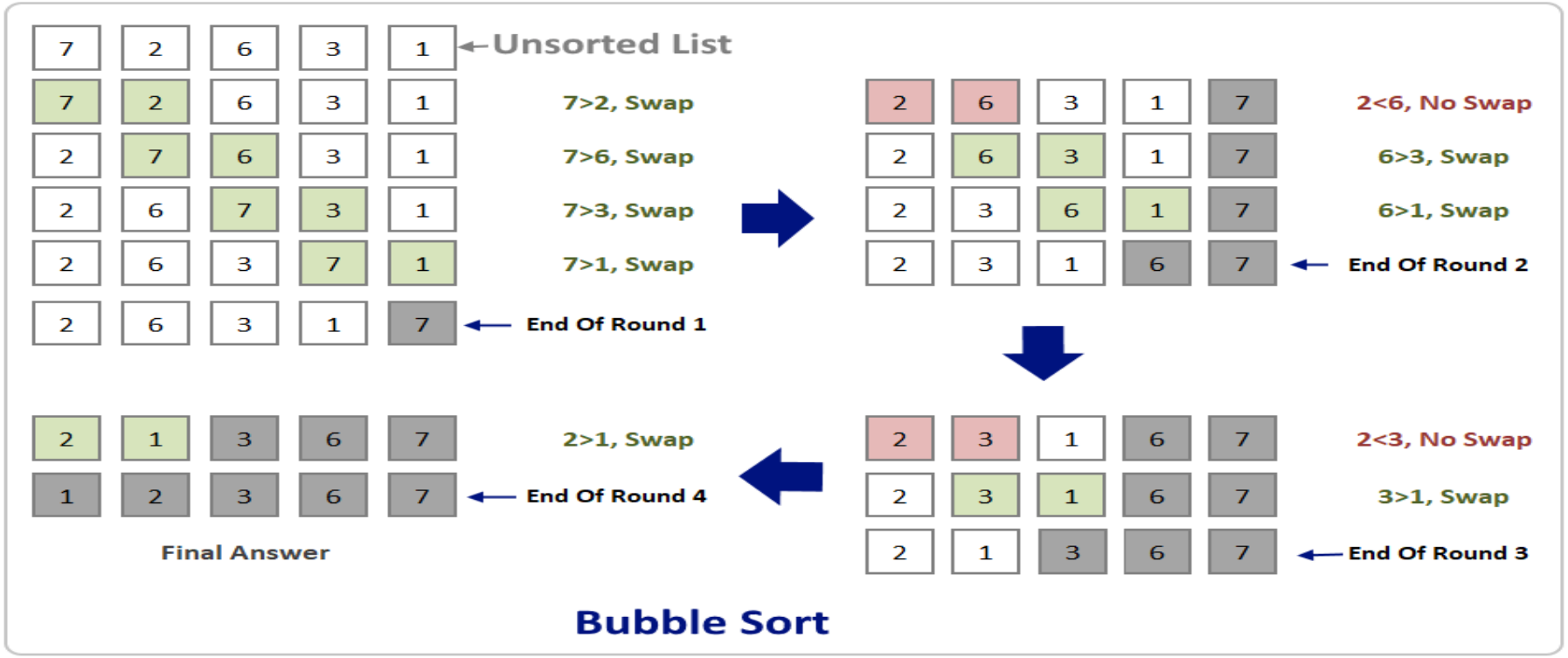
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Heap Sort

1. Bubble Sort

Bubble sort is a type of sorting.

It is used for sorting 'n' (number of items) elements.

It compares all the elements one by one and sorts them based on their values.



2. Insertion Sort

- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- Insertion sort has one of the simplest implementation.
- This sort is efficient for smaller data sets but it is insufficient for larger lists.
- It has less space complexity like bubble sort.
- It requires single additional memory space.
- Insertion sort does not change the relative order of elements with equal keys because it is stable.

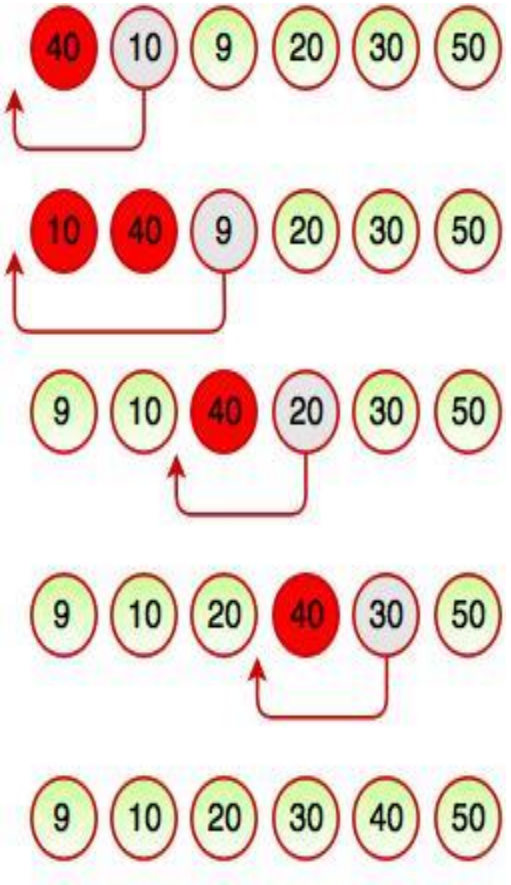


Fig. Working of Insertion Sort

The above diagram represents how insertion sort works. Insertion sort works like the way we sort playing cards in our hands. It always starts with the second element as key. The key is compared with the elements ahead of it and is put it in the right place.

In the above figure, 40 has nothing before it. Element 10 is compared to 40 and is inserted before 40. Element 9 is smaller than 40 and 10, so it is inserted before 10 and this operation continues until the array is sorted in ascending order.

Selection Sort

Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position. Then finds the second smallest element and exchanges it with the element in the second position and continues until the entire array is sorted.

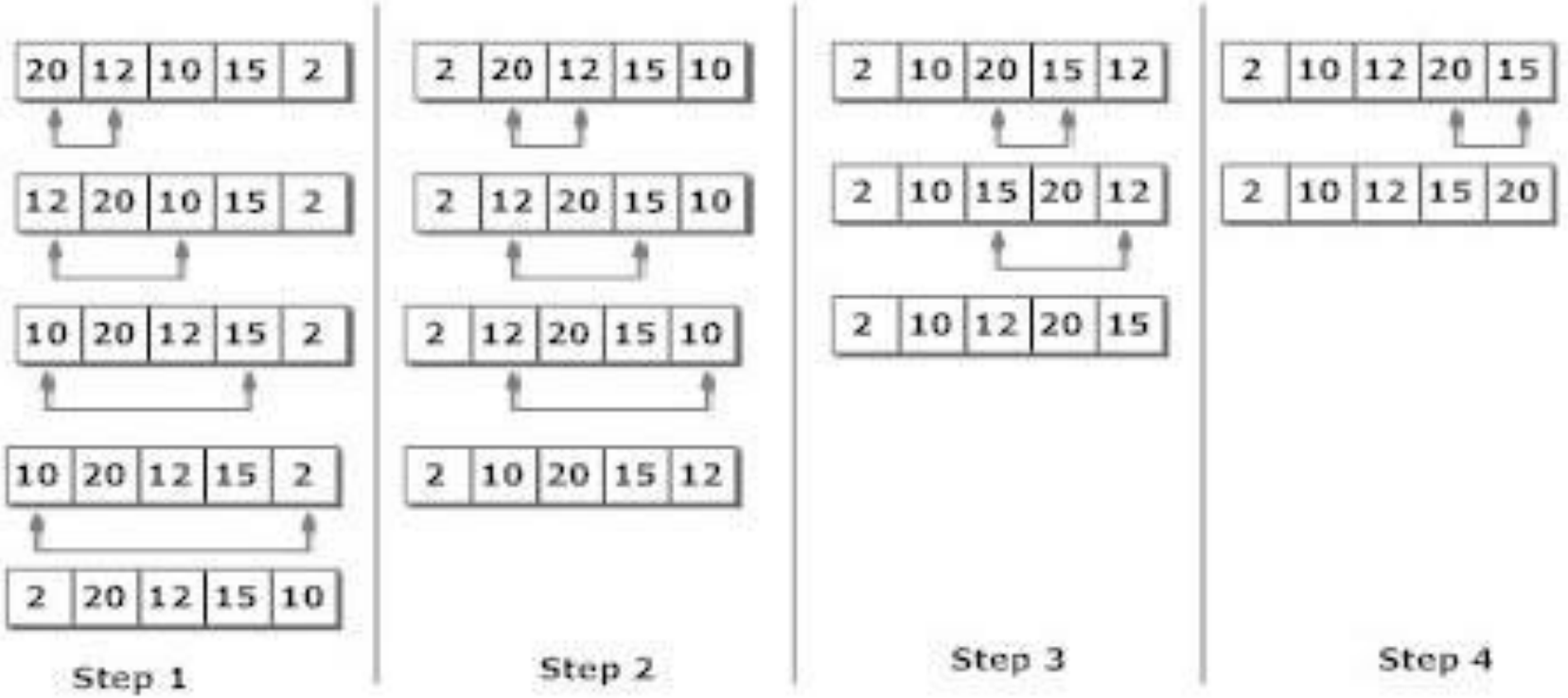
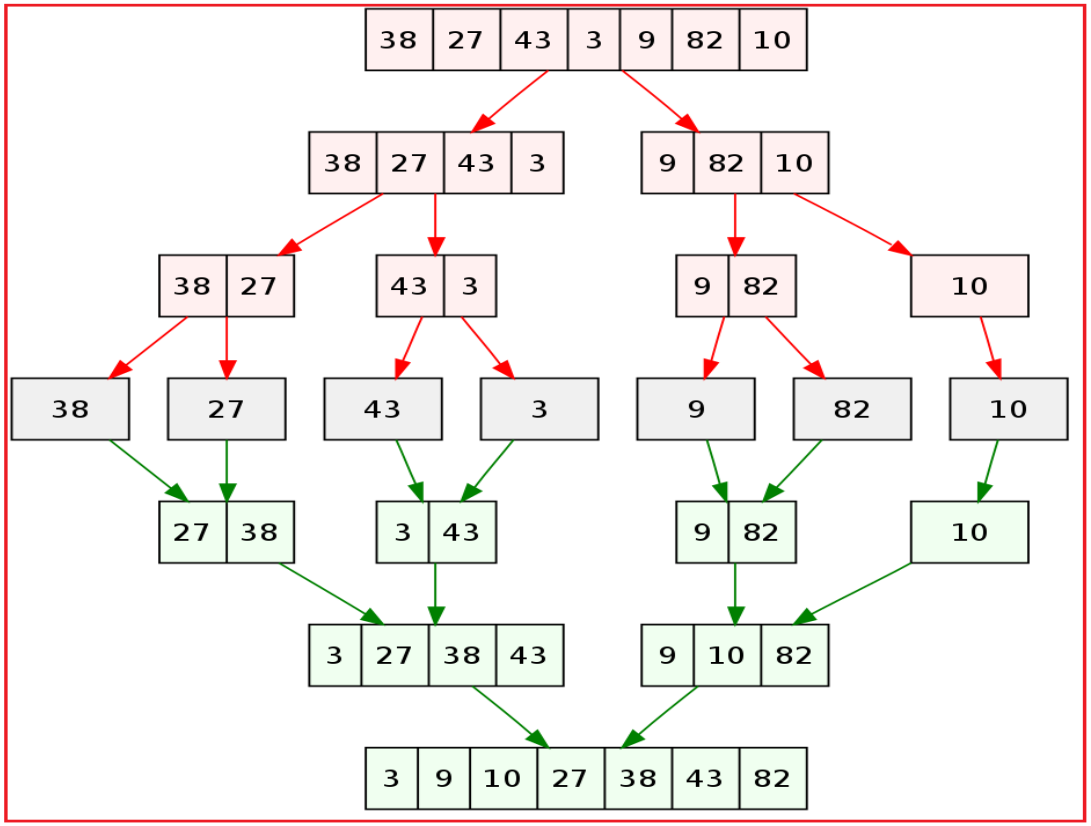


Figure: Selection Sort

Merge sort

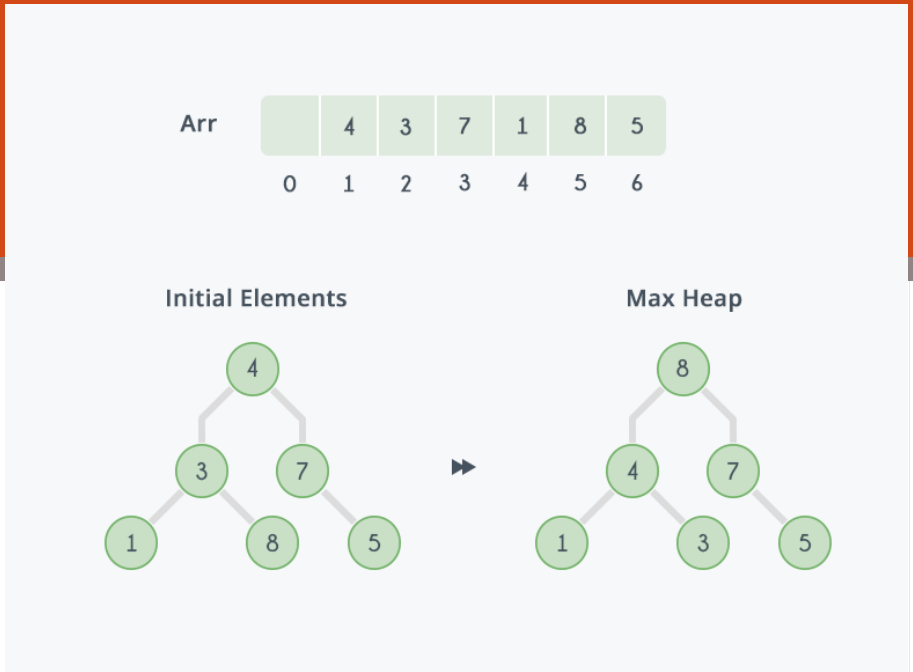
Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

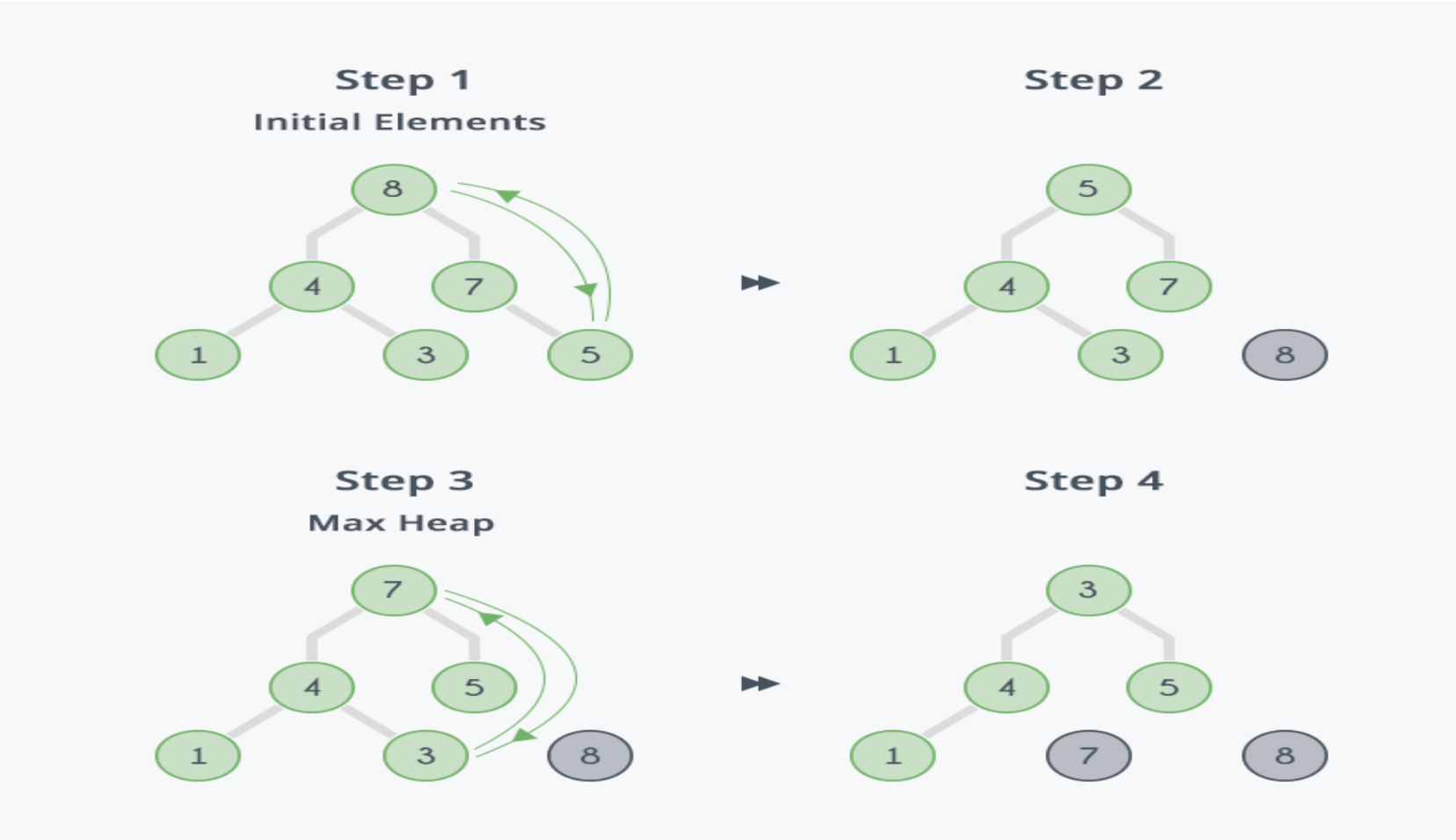


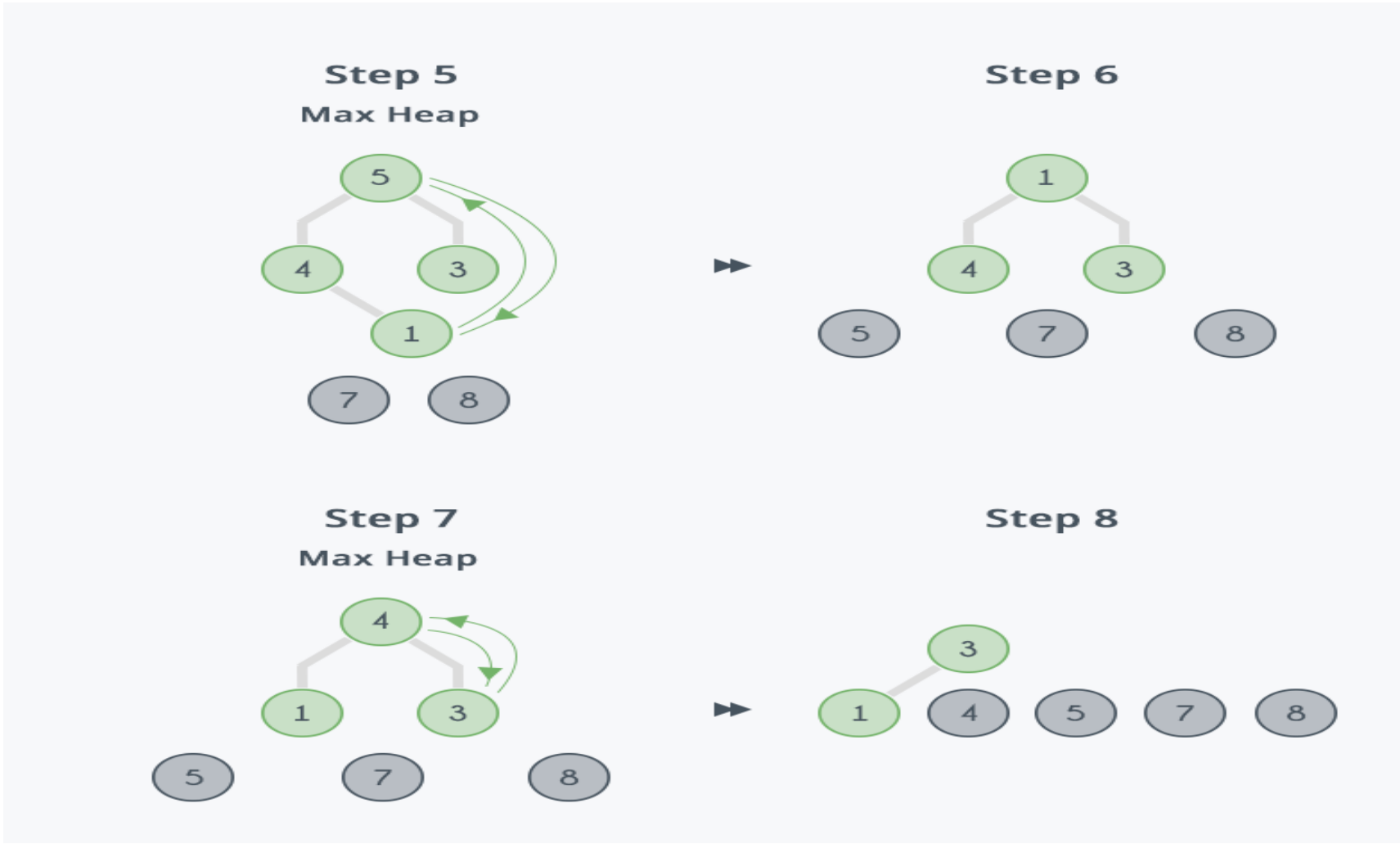
Heap Sort

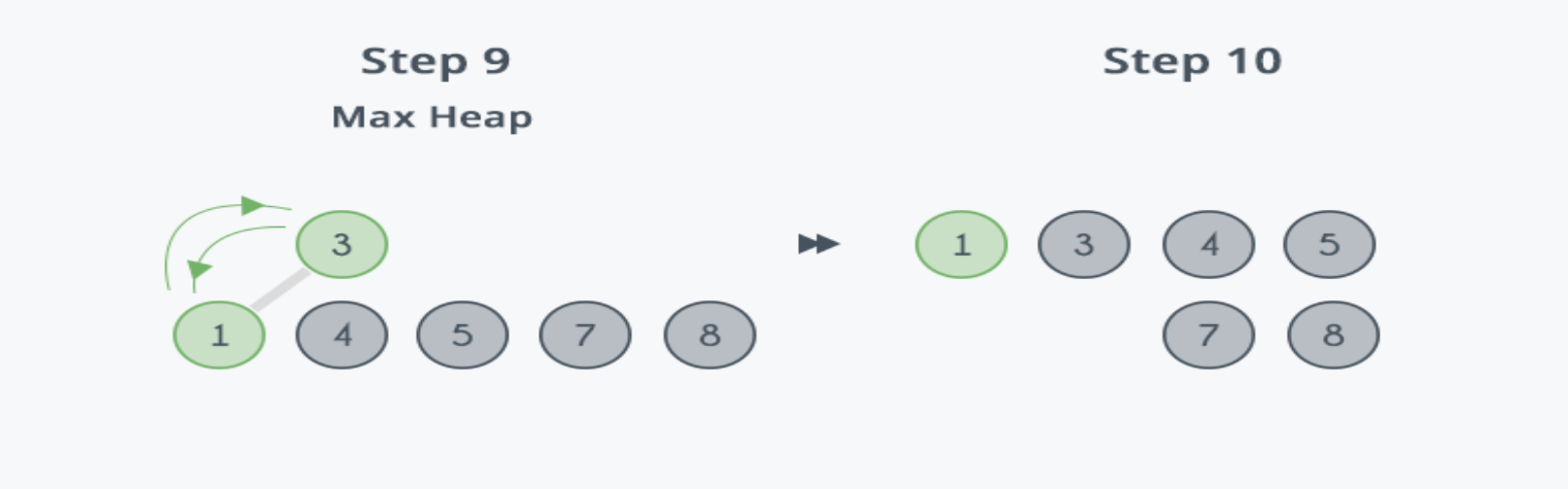
Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.

Step 1: 8 is swapped with 5.
Step 2: 8 is disconnected from heap as 8 is in correct position now and.
Step 3: Max-heap is created and 7 is swapped with 3.
Step 4: 7 is disconnected from heap.
Step 5: Max heap is created and 5 is swapped with 1.
Step 6: 5 is disconnected from heap.
Step 7: Max heap is created and 4 is swapped with 3.
Step 8: 4 is disconnected from heap.
Step 9: Max heap is created and 3 is swapped with 1.
Step 10: 3 is disconnected.




Step 1
Initial Elements
8
4
7
1
3
5
Step 2
5
4
7
1
3
8
Step 3
Max Heap
7
4
5
1
3
8
Step 4
3
4
5
1
7
8


Step 5
Max Heap
5
4
3
1
7
8
Step 6
1
4
3
5
7
8
Step 7
Max Heap
4
1
3
5
7
8
Step 8
3
1
4
5
7
8



After all the steps, we will get a sorted array.

Arr		1	3	4	5	7	8
	0	1	2	3	4	5	6

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8<u>2</u>, 90<u>1</u>, 10<u>0</u>, 1<u>2</u>, 15<u>0</u>, 7<u>7</u>, 5<u>5</u> & 2<u>3</u>

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Pass - 1

Pass - 2

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1<u>0</u>0, 1<u>5</u>0, 9<u>0</u>1, <u>8</u>2, <u>1</u>2, <u>2</u>3, <u>5</u>5 & <u>7</u>7

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Pass - 3

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

<u>1</u>00, <u>9</u>01, 12, 23, <u>1</u>50, 55, 77 & 82

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. This algorithm follows divide and conquer approach.

A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets.

Quick Sort : Rules

Rule 1 : Before start the sortingMake first element as Pivot
Second element as P and
Last element as Q

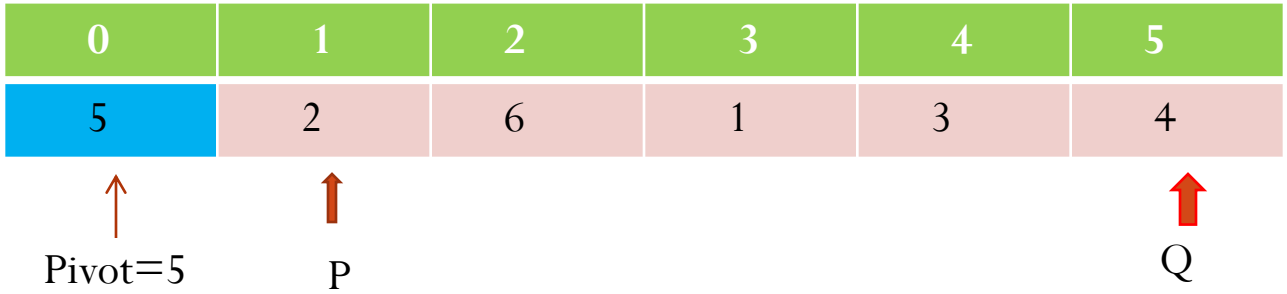
Rule 2 : Check if P> Pivot,
If no then move P to right.
Other wise swap P and Q

Rule 3 : Check if Q< Pivot,
If no then move to Q left
Other wise swap P and Q

Rule 4 :Repeat same process till P and Q crosses with each other
if P and Q crosses with each other then swap Q and Pivot element
and break the list in two parts and
repeat same process for both the list until you get sorted array

Quick Sort

Array



Step : 1

Check if P> Pivot,

If no then move to right other wise swap P and Q

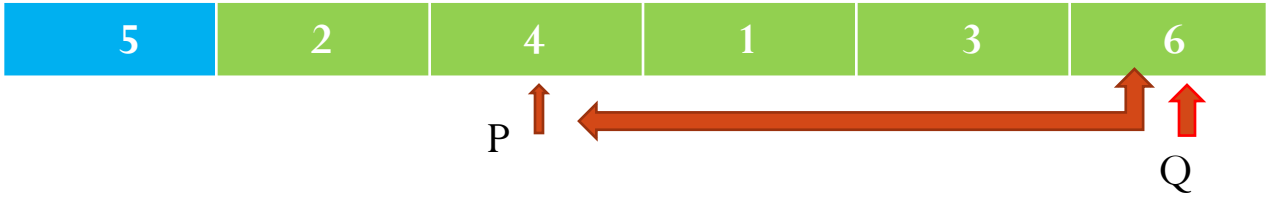
so here in example 2>5 no then move to right after that p comes at pace 6

Check if Q< Pivot,

If no then move to left other wise swap P and Q

Here in example 4<5 yes so swap P and Q ,then swap 6 and 4

Step : 2

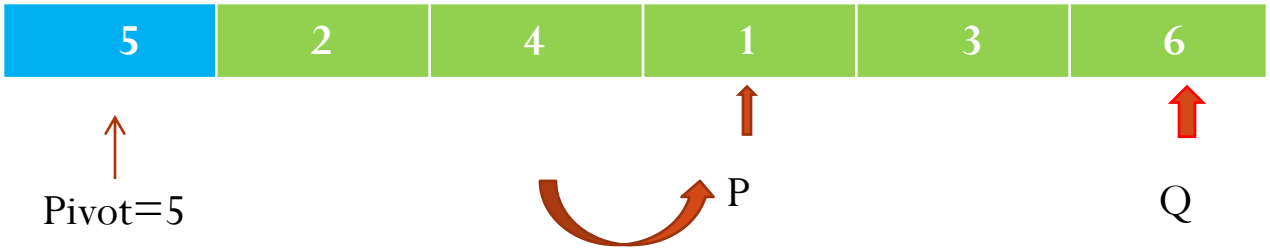


Then again repeat same process till P and Q crosses with each other

Step : 5

i.e. Check if P> Pivot, If no then move to right other wise swap P and Q

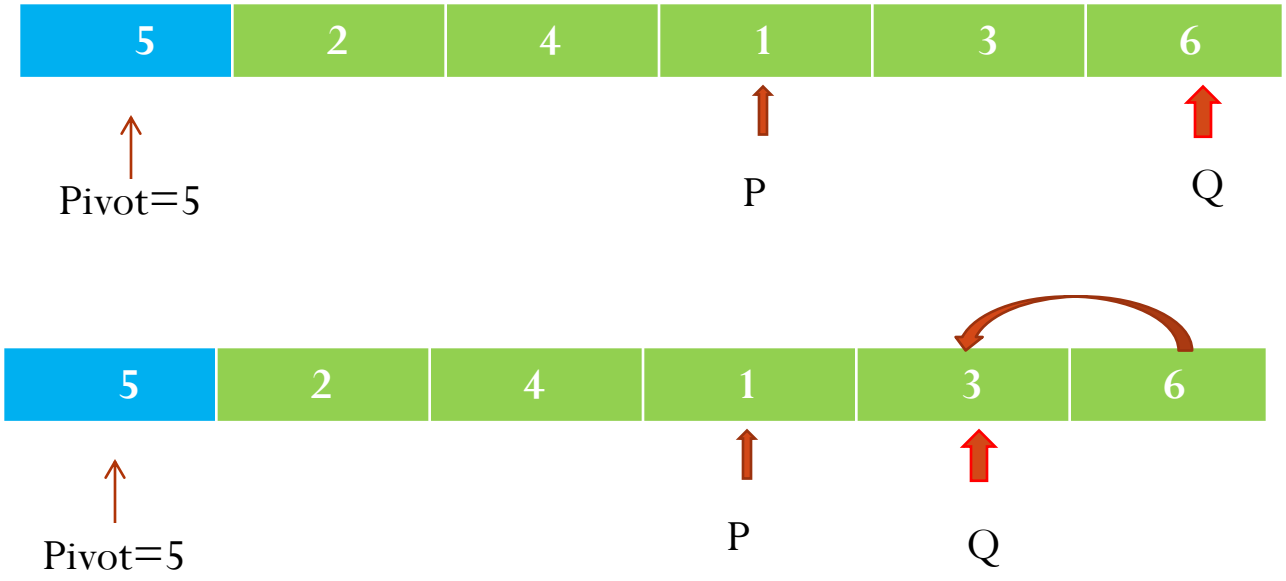
Here in example 6<5 no then move to left after that Q comes at pace 3



Step : 4

Check if Q< Pivot, If no then move to left other wise swap P and Q

Here in example 6<5 no then move to left after that Q comes at pace 3

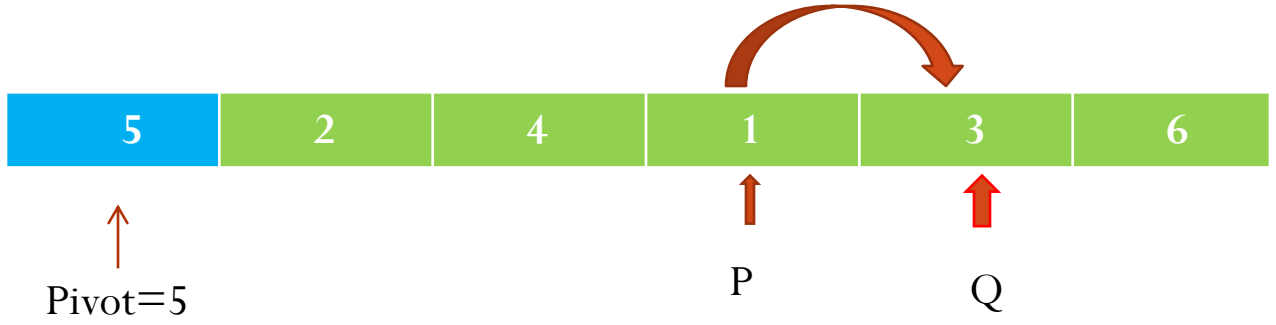


Step : 5

Then again repeat same process till P and Q crosses with each other

i.e. Check if P> Pivot, If no then move to right other wise swap P and Q

so here in example 1>5 no then move to right but while moving it crosses with Q



Step : 6

Then swap pivot with Q so it becomes and break the list in two parts

So lists became

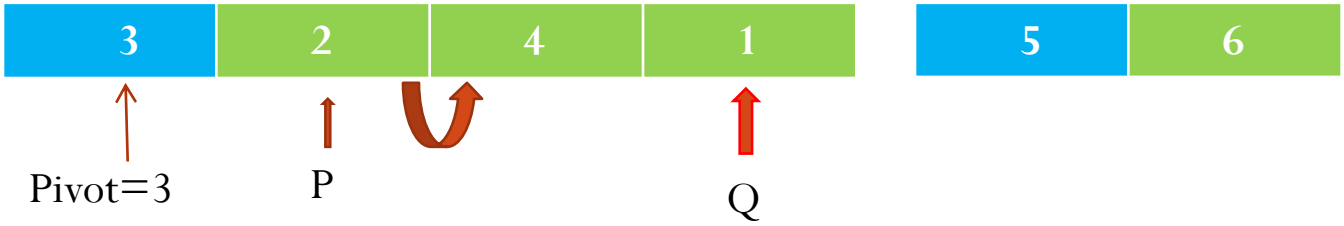
Apply same method to the left side elements

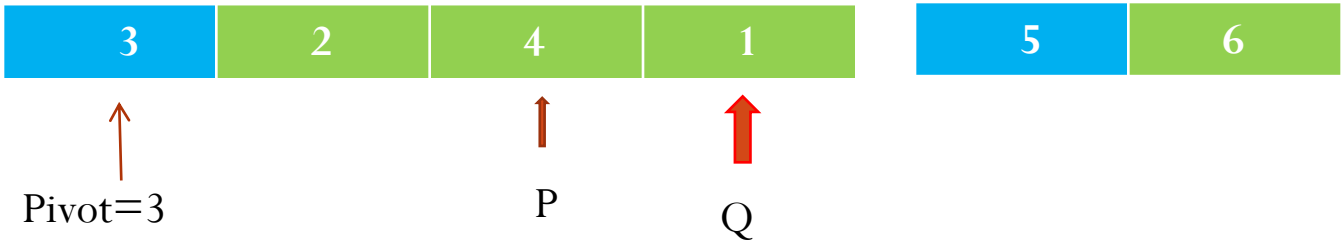
Step : 7

Check if P> Pivot,

If no then move to right other wise swap P and Q

so here in example 2>3 no then move to right after that p comes at pace 4





Step : 8

Check if Q< Pivot,

If no then move to left other wise swap P and Q

Here in example 1<3 yes so swap P and Q ,but while swapping p and Q crosses so swap Q with pivot as per the rule

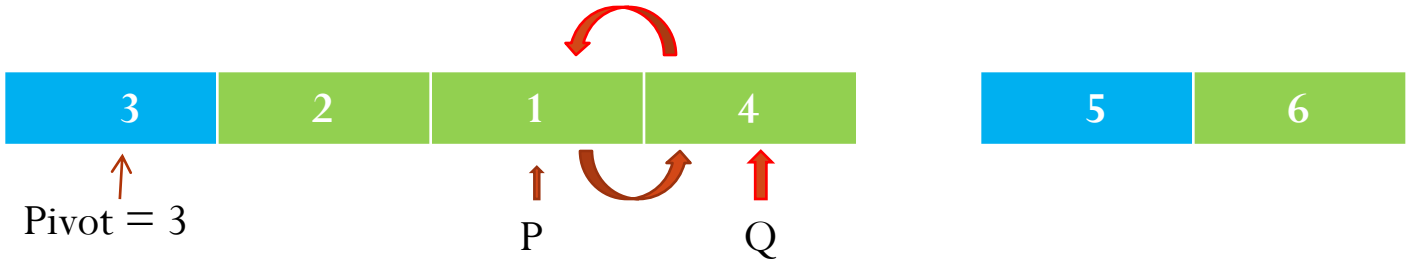
Step : 8

Check if Q< Pivot,

If no then move to left other wise swap P and Q

Here in example 1<3 yes so swap P and Q ,

but while swapping p and Q crosses so swap Q with pivot as per the rule



After swapping the left list becomes and it also get sorted .

Also we have to simultaneously we have to apply same process of right hand side list but in right side list there is only two element and they are in sorted order

So after combing both the list we get sorted array as below

UNIT 3 : Linked list

3.2 Implementation of Linked List – Static & Dynamic representation,

3.3 Types of Linked List

- Singly Linked list(All type of operation)
- Doubly Linked list (Create , Display)
- Circularly Singly Linked list (Create, Display)
- Circularly Doubly Linked list (Create, Display)

3.4 Generalized linked list – Concept and Representation

What is Linked List?

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data.

The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence.

Each element in a linked list is called "Node".

What is Single Linked List?

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

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

In any single linked list, the individual element is called as "Node".

Every "Node" contains two fields, data field, and the next field.

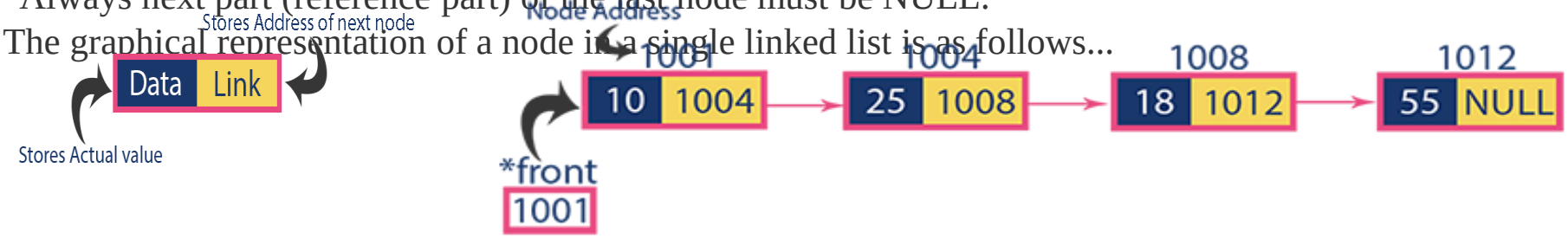
The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.

Important Points to be Remembered

In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").

Always next part (reference part) of the last node must be NULL.

The graphical representation of a node in a single linked list is as follows...



Operations on Single Linked List

The following operations are performed on a Single Linked List

- **Insertion**
- **Deletion**
- **Display**

Inserting node at Beginning of the list

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

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

1. Inserting node at Beginning of the list

```
void insertAtBeginning(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(
struct Node));
  newNode->data = value;
  if(head == NULL)
  {
    newNode->next = NULL;
    head = newNode;
  }
  else
  {
    newNode->next = head;
    head = newNode;
  }
  cout<<"One node inserted!!!";
}
```

Inserting node At End of the list

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

- **Step 1 -** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- **Step 6 -** Set **temp** → **next** = **newNode**.

2. Inserting node At End of the list

```
void insertAtEnd(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof
(struct Node));
  newNode->data = value;
  newNode->next = NULL;
  if(head == NULL)
        head = newNode;
  else
  {
    struct Node *temp = head;
    while(temp->next != NULL)
        temp = temp->next;
    temp->next = newNode;
  }
  cout<<"One node inserted!!!";
}
```

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

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

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode** → **next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode** → **next** = **temp** → **next**' and '**temp** → **next** = **newNode**'

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

```
void insertBetween(int value, int loc1, int loc2
)

{  struct Node *newNode;

  newNode = (struct Node*)malloc(sizeof(stru
ct Node));

  newNode->data = value;

  if(head == NULL)

  {  newNode->next = NULL;

    head = newNode;

  }

  else

  { struct Node *temp = head;

    while(temp->data != loc1 && temp->data !
= loc2)

    temp = temp->next;

    newNode->next = temp->next;

    temp->next = newNode;

  }

  cout<<"One node inserted!!!;
```

Deletion

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

1.Deleting from Beginning of the list

2.Deleting from End of the list

3.Deleting a Specific Node

Deleting from Beginning of the list

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

•**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

•**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

•**Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

•**Step 4 -** Check whether list is having only one node (**temp → next** == **NULL**)

•**Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list

Deleting from Beginning of the list

```
void removeBeginning()

{

  if(head == NULL)

          printf("\n\nList is Empty!!!")
;

  else

  {       struct Node *temp = head;

    if(head->next == NULL)

    {    head = NULL;  //stores null val
ue

                              in  head  pointe
r

        free(temp);

    }

    else

    {     head = temp->next;

          free(temp);

          printf("\nOne node deleted!!!
\n\n");
```

Deleting from End of the list

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

•**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

•**Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.

•**Step 3 -** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.

•**Step 4 -** Check whether list has only one Node (**temp1** → **next** == **NULL**)

•**Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)

•**Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1** → **next** == **NULL**)

•**Step 7 -** Finally, Set **temp2** → **next** = **NULL** and delete **temp1**.

Deleting node from End of the list

```
void removeEnd()
{       if(head == NULL)
   {    printf("\nList is Empty!!!\n");
   }
   else
   {       struct Node *temp1 = head,*te
mp2;
      if(head->next == NULL)
            head = NULL;
      else
      {    while(temp1->next != NULL)
            {   temp2 = temp1;
                temp1 = temp1->next;
            }
            temp2->next = NULL;
      }
      free(temp1);
      printf("\nOne  node  deleted!!!\n\n
");
```

Deleting a Specific Node from the list

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

•**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

•**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

•**Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

•**Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

•**Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

•**Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

•**Step 7 -** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

•**Step 8 -** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).

Deleting a Specific Node from the list

```
void removeSpecific(int delValue)
{
struct Node *temp1 = head, *temp2;
while(temp1->data != delValue)
{
if(temp1 -> next == NULL)
{
cout<<"Given node not found in the list!!!";
goto functionEnd;
}
temp2 = temp1;
temp1 = temp1 -> next;   }
temp2 -> next = temp1 -> next;
free(temp1);
cout<<"One node deleted!!!";
functionEnd:
}
```

Displaying a elements in Single Linked List

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

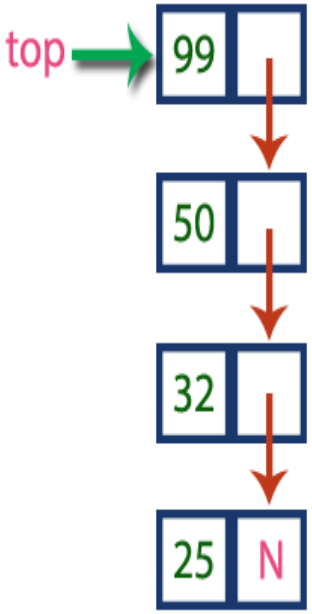
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying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

Displaying a Single Linked List

```
void display()
{
 if(head == NULL)
{
 cout<<List is Empty\n");
}
 else
{
struct Node *temp = head;
 cout<<"List elements are ";
 while(temp->next != NULL)
 {
 cout<<temp->data;
 temp = temp->next;
 }
   cout<<temp->data;
 }}
```

Stack Using Linked List

stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.




```
struct Node
{
    int data;
    struct   Node
*next;
}*top = NULL;
```

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

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

Step 2 - Check whether stack is **Empty** (**top** == **NULL**)

Step 3 - If it is **Empty**, then set **newNode** → **next** = **NULL**.

Step 4 - If it is **Not Empty**, then set **newNode** → **next** = **top**.

Step 5 - Finally, set **top** = **newNode**.

push() function :

```
void push(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  if(top == NULL)
   {
        newNode->next = NULL;
   }
  else
   {
        newNode->next = top;
   }
  top = newNode;
  printf("\nInsertion is Success!!!\n");
}
```

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

Step 1 - Check whether **stack** is **Empty** (**top == NULL**).

Step 2 - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

Step 3 - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

Step 4 - Then set '**top** = **top** → **next**'.

Step 5 - Finally, delete '**temp**'. (**free(temp)**).

pop() function :

```
void pop()
{
if(top == NULL)
{
        printf("\nStack is Empty!!!\n");
}
else
{
        struct Node *temp = top;
}
 printf("\nDeleted element: %d", temp->data);
top = temp->next;
free(temp);
 }
```

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

Step 1 - Check whether stack is **Empty** (**top** == **NULL**).

Step 2 - If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.

Step 4 - Display '**temp** → **data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp** → **next** != **NULL**).

Step 5 - Finally! Display '**temp** → **data** ---> **NULL**'.

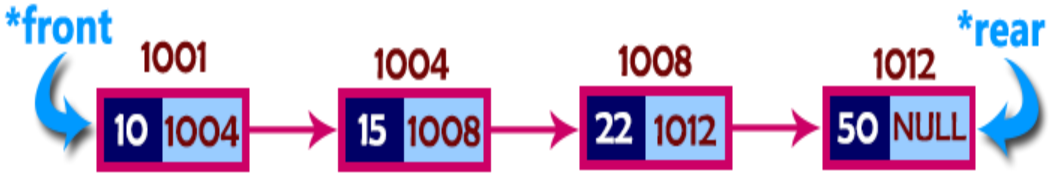
display() Function :

```
void display()
{
  if(top == NULL)
{
    printf("\nStack is Empty!!!\n");
}
  else
{
    struct Node *temp = top;
}
    while(temp->next != NULL)
{
        printf("%d--->",temp->data);
        temp = temp -> next;
 }
    printf("%d--->NULL",temp->data);
}
```

QUEUE Using Linked List

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

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



QUEUE Using Linked List

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

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

```
struct Node
{
int data;
 struct Node *next;
}*front = NULL,*rear =
NULL;
```

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

•**Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.

•**Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)

•**Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.

•**Step 4 -** If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

Insert () Function :

```
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode -> next = NULL;
  if(front == NULL)
{
    front = rear = newNode;
}
  else
{
    rear -> next = newNode;
    rear = newNode;
  }
  printf("\nInsertion is Success!!!\n");
}
```

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

Step 1 - Check whether **queue** is **Empty** (**front == NULL**).

Step 2 - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function

Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.

Step 4 - Then set '**front** = **front** → **next**' and delete '**temp**' (**free(temp)**).

delete() Function:

```
void delete()
{
  if(front == NULL)
{
    printf("\nQueue is Empty!!!\n");
}
  else
{
    struct Node *temp = front;
}
    front = front -> next;
    printf("\nDeleted element: %d\n", te
mp->data);
    free(temp);
  }
}
```

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2 -** If it is **Empty** then, display '**Queue is Empty!!!**' and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **front**.
- **Step 4 -** Display '**temp** → **data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp** → **next** != **NULL**).
- **Step 5 -** Finally! Display '**temp** → **data** ---> **NULL**'.

display() Function:

```
void display()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    while(temp->next != NULL){
        printf("%d--->",temp->data
);
        temp = temp -> next;
    }
    printf("%d--->NULL\n",temp->
data);
  }
}
```

UNIT 4: Stack

Stacks

Introduction and Definition

Representation of Stacks

Operations on Stacks

Applications of Stacks

Representation of Arithmetic Expressions

Infix

Postfix

Prefix

Unit 2. Stack

What is Stack?

- Stack is an ordered list of the same type of elements.
- It is a linear list where all insertions and deletions are permitted only at one end of the list.
- Stack is a LIFO (Last In First Out) structure.
- In a stack, when an element is added, it goes to the top of the stack.

Definition

"Stack is a collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Operations on Stack

1. **CREATE**
2. **PUSH**
3. **POP**
4. **ISEMPTY**
5. **ISFULL**
6. **DISPLAY/TRAVERSE**

Operation on Stack

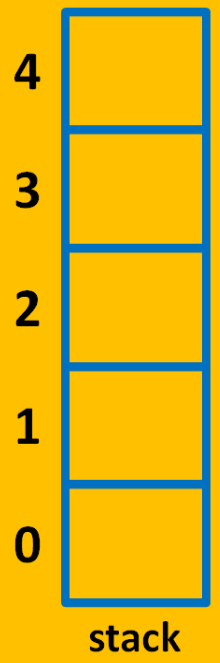
1. Create

   This operation create a stack, which is empty

Operations on STACK ?

- **Creation**
- Insertion
- Deletion
- Displaying

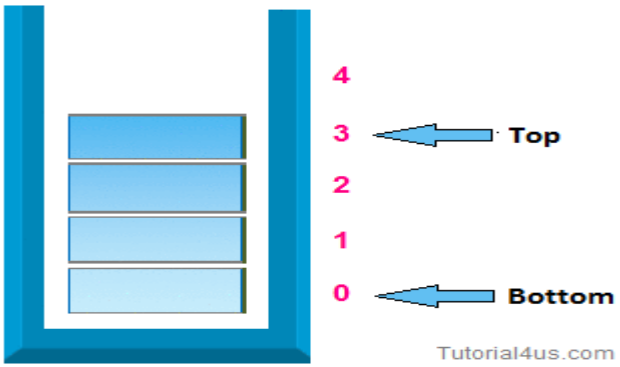
```
#define   SIZE  5
int stack[SIZE];
```



Operation on Stack

2. Push

The push operation adds a new element to the stack. As stated above, any element added to the stack goes at the top, so push adds an element at the top of a stack



push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack…

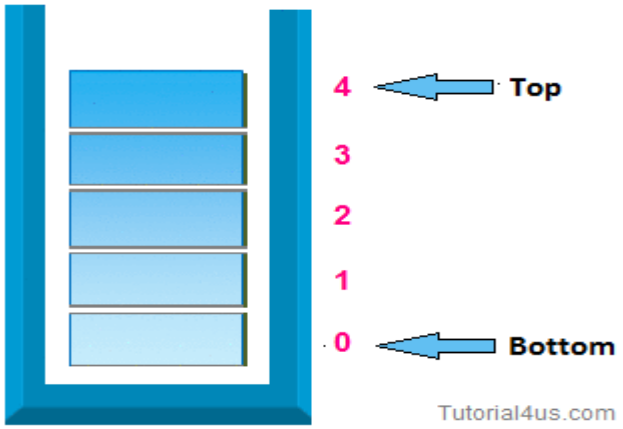
Step 1 - Check whether **stack** is **FULL**. (**top == SIZE-1**)

Step 2 - If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

3 Pop

The pop operation removes and also returns the top-most (or most recent element) from the stack.



pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack…

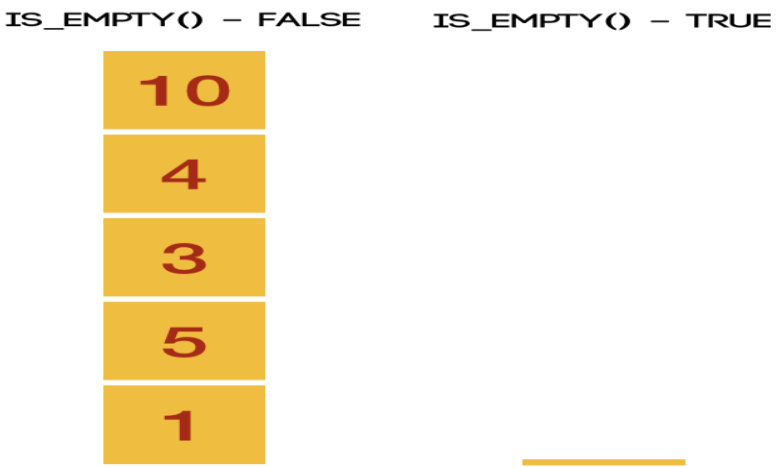
Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)

Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

3. isEmpty()

- This operation checks whether a stack is empty or not i.e., if there is any element present in the stack or not.
- When a stack is completely full, it is said to be Overflow state and if stack is completely empty, it is said to be Underflow state.



4. Isfull()

This operation checks whether the stack isfull. It returns TRUE if stack is full and false otherwise

5. display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack…

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)

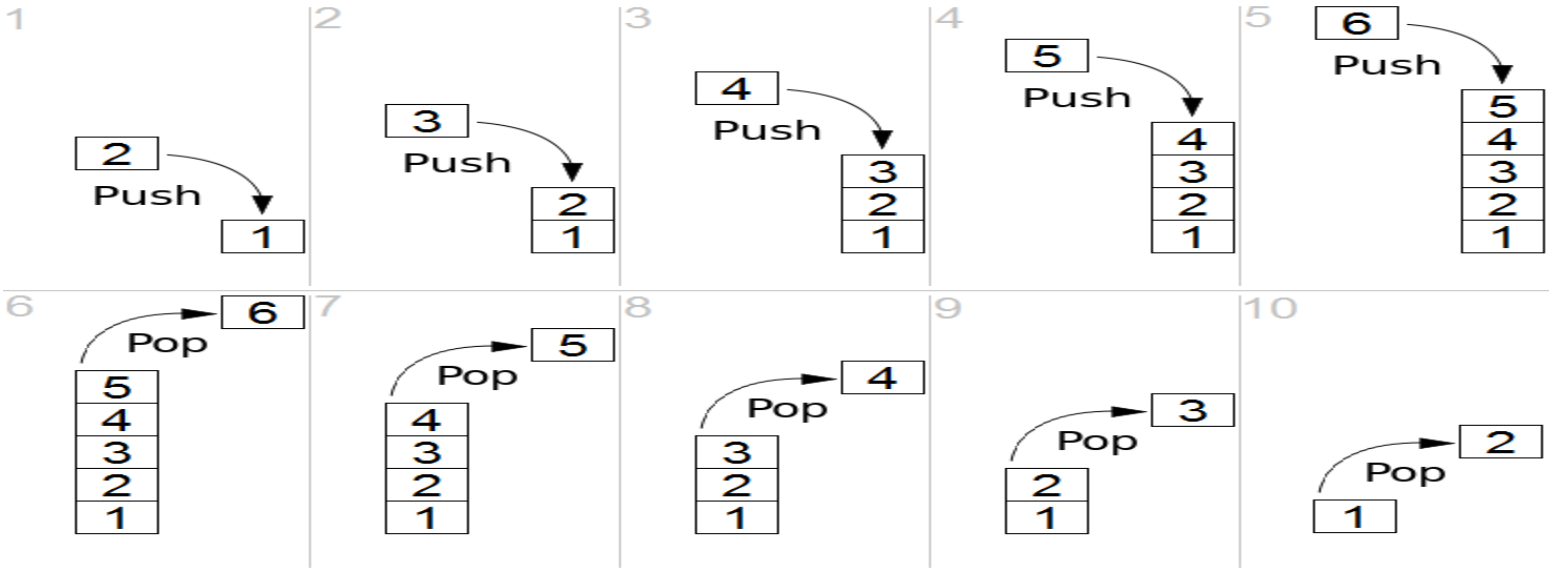
Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).

Step 4 - Repeat above step until **i** value becomes '0'.

Implementation of Stack (Static Implementaton)

- The below diagram represents a stack insertion and deletion operation.
- In a stack, inserting and deleting of elements is performed at a single position which is known as, **Top**.
- Insertion operation can be performed using Push() function and deletion operation can be performed using Pop() function .
- New element is added and deleted at top of the stack
- Delete operation is based on LIFO principle.



Following table shows the Position of Top which indicates the status of stack

Position of Top	Status of Stack
-1	Stack is empty.
0	Only one element in a stack.
N - 1	Stack is full.
N	Stack is overflow. (Overflow state)

Applications of Stack :

Following are some of the important applications of a Stack data structure:

1. Stacks can be used for expression evaluation.
2. Stacks can be used to check parenthesis matching /correctness of nested parenthesis.
3. Reversing a string.
4. Check whether string is palindrome or not.
5. Stacks can be used for Conversion from one form of expression to another.

   A. Infix to Postfix or Infix to Prefix Conversion

   The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent.

   B. Postfix or Prefix Evaluation

   These postfix or prefix notations are used in computers to express some expressions.
6. Stacks can be used for Memory Management.
7. Stack data structures are used in backtracking problems.

   Backtracking *can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.*

backtracking is, that is solving all sub-problems one by one in order to reach the best possible solution.

Decision Problem – In this, we search for a feasible solution.
Optimization Problem – In this, we search for the best solution.
Enumeration Problem – In this, we find all feasible solutions.

Functions of Stack

Push()	Pop()	Display()

void push()
{
int val;
if(top==MAX-1)
{
printf("\nStack is full!!");
}
else
{
printf("\nEnter element to push:");
scanf("%d",&val);
top=top+1;
stack[top]=val;
}
}
	void pop()
	{
	if(top==-1)
	{
	printf("\nStack is empty!!");
	}
	else
	{
	printf("\nDeleted element is %d",stack[top]);
	top=top-1;
	}
	}
		void display()
		{
		int i;
		if(top == -1)
		{
		printf("\n\n Stack is Empty.");
		}
		else
		{
		for(i=top; i>=0; i--)
		{
		printf("\n%d", stack[i]);
		}
		}
		
		}

Prog. Stack using Array

```
#include <stdio.h>
#include <conio.h>
#define MAX 50
void push();
void pop();
void display();
int stack[MAX], top=-1, element;
void main()
{
   int ch;
   do
   {
      printf(" 1. push\n 2. pop 3. Display\n 4. Exit\n");
      printf("\n Enter Your Choice: ");
      scanf("%d", &ch);
      switch(ch)
      {
         case 1:
         push();
         break;
         case 2:
         pop();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(0);
         default:
         printf("\n\n Invalid entry try again...\n");
      }
   }while(ch!=4);
   getch(); }
```

```
void push()
{
   if(top == MAX-1)
      printf("\n\n Stack is Full.");
   else
   {
      printf("\n\n Enter Element: ");
      scanf("%d", &element);
      top++;
      stack[top] = element;
  printf("\n\n Element Inserted = %d", element);
   }
}
void pop()
{
   if(top == -1)
      printf("\n\n Stack is Empty.");
   else
   {
      element = stack[top];
      top--;
   printf("\n\n Element Deleted = %d", element);
   }
}
void display()
{
   int i;
   if(top == -1)
   printf("\n\n Stack is Empty.");
   else
   {
      for(i=top; i>=0; i--)
         printf("\n%d", stack[i]);
   }
}
```

1.push/Insert

```
1. Insert
2. Delete
3. Display
4. Exit

Enter Your Choice: 1

Enter Element: 30

Element Inserted = 30


1. Insert
2. Delete
3. Display
4. Exit

Enter Your Choice:
```

2.Display

```
1. Insert
2. Delete
3. Display
4. Exit

Enter Your Choice: 3

30

20

10
```

3.pop/Delete

```
1. Insert
2. Delete
3. Display
4. Exit

Enter Your Choice: 2

Element Deleted = 30


1. Insert
2. Delete
3. Display
4. Exit

Enter Your Choice: 3

20

10
```

ARRAY VERSUS STACK

ARRAY	STACK
A data structure consisting of a collection of elements each identified by the array index	An abstract data type that serves as a collection of elements with two principal operations: push and pop
Contains elements of the same data type	Contains elements of different data types
Basic operations include insert, delete, modify, traverse, sort, search and merge	Basic operations are push, pop and peek
Any element can be accessed using the array index	Only the topmost element can be read or removed at a time

Expression Evaluation and Conversion

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: (A + B) * (C – D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as suffix notation and is also referred to reverse polish notation.

Example: A B + C D – *

Infix	Prefix	Postfix
a + b	+ b a	a b +
(a + b) * (c + d)	* + d c + b a	a b + c d + *
b * b - 4 * a * c	- * c * a 4 * b b	b b * 4 a * c * -

Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. 1. Scan the infix expression from left to right.
2. If the scanned symbol is left parenthesis, push it onto the stack.
3. If the scanned symbol is an operand, then place directly in the postfix expression (output).
4. If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
5. If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Conversion of Infix to Postfix

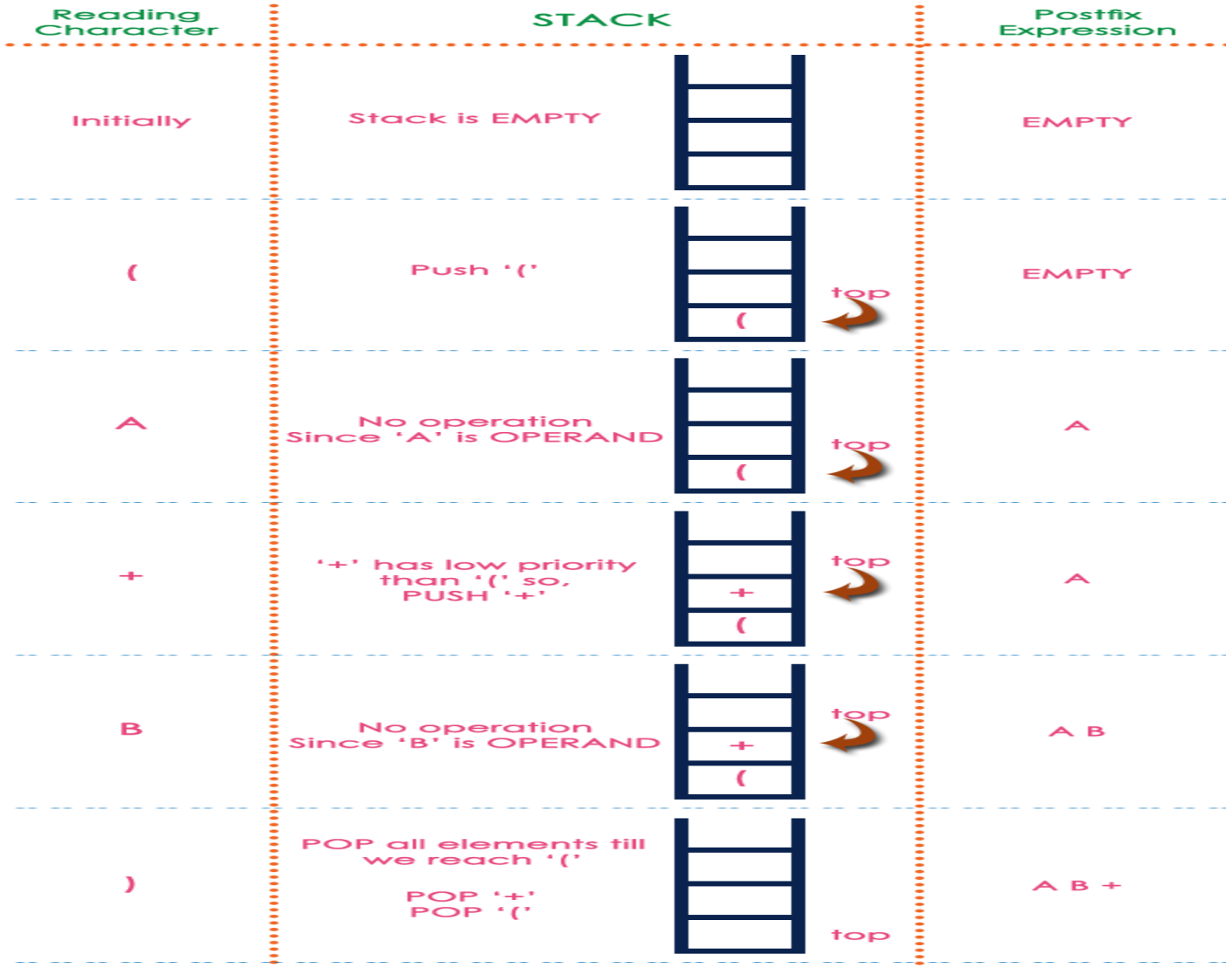
Example to Convert Infix to Postfix using stack

a + (b*c)

Read character	Stack	Output
a	Empty	a
+	+	a
(	+(	a
b	+(	ab
*	+(*	ab
c	+(*	abc
)	+	abc*
		abc*+

Example : (A + B) * (C - D)

The given infix expression can be converted into postfix expression using Stack data Structure as follows...



*	Stack is EMPTY & '*' is Operator PUSH '*'		A B +
(	PUSH '('		A B +
C	No operation Since 'C' is OPERAND		A B + C
-	'-' has low priority than '(' so, PUSH '-'		A B + C
D	No operation Since 'D' is OPERAND		A B + C D
)	POP all elements till we reach '(' POP '-' POP '('		A B + C D -
$	POP all elements till Stack becomes Empty		A B + C D - *

1. Infix to Postfix Conversion

Following table shows the evaluation of Infix to Postfix:

Example: Suppose we are converting A*B/(C-D)+E*F expression into postfix form.

Input	Stack	Output
A	Empty	A
*	*	A
B	*	AB
/	/	AB*
(	/(	AB*
C	/(	AB*C
-	/(-	AB*C
D	/(-	AB*CD
)	-	AB*CD-
+	+	AB*CD-/
E	+	AB*CD-/E
*	+*	AB*CD-/E
F	+*	AB*CD-/EF
	Empty	**AB*CD-/EF*+**

So, the Postfix Expression is AB***CD-/EF*+**

Infix to Prefix Conversion

Example : (a+b)*(c+d)

Reverse expression : (d+c)*(b+a)

Input Character	Stack	Output
(	(	
d	(	d
+	(+	d
c	(+	dc
)		dc+
*	*	dc+
(	*(	dc+
b	*(	dc+b
+	*(+	dc+b
a	*(+	dc+ba
)	*(+	dc+ba
	*	dc+ba+
Final Prefix Expression is :		dc+ba+*
After reversing final Prefix Expression is :		*+ab+cd

Following table shows the evaluation of Infix to Prefix:

Conversion from infix to prefix:

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression A + B – C into prefix expression

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example convert Infix expression to prefix expression = **(A+B^C)*D+E^5**

Step 1. Reverse the infix expression.

5^E+D*)C^B+A(

Step 2. Make Every '(' as ')' and every ')' as '('

5^E+D*(C^B+A)

Step 3. Convert expression to postfix form.

A+(B*C-(D/E-F)*G)*H

Step 4. Reverse the expression.

+*+A^BCD^E5

Input Character	Stack	Output
5	Empty	-
^	Empty	5
E	^	5
+	^	5E
D	+	5E^
*	+	5E^D
(	+*	5E^D
C	+*(	5E^D
^	+*(	5E^DC
B	+*(^	5E^DC
+	+*(^	5E^DCB
A	+*(+	5E^DCB^
)	+*(+	5E^DCB^A
End	+*	5E^DCB^A+
End	Empty	5E^DCB^A+*+

Result

+*+A^BCD^E5

4. Prefix to Infix

Example: Convert Prefix to Infix /-bc+-pqr

Input Character	Stack
r	Empty
q	"r"
p	"q" "r"
-	"p" "q" "r"
+	"p-q" "r"
c	"p-q+r"
b	"c" "p-q+r"
-	"b" "c" "p-q+r"
/	"b-c" "p-q+r"
NULL	**"((b-c)/((p-q)+r))"**

So, the Infix Expression is
((b-c)/((p-q)+r))

4. Postfix to to Infix

Example: Convert Prefix to Infix ab-de+f*/

Input Character	Stack
a	Empty
b	"a"
-	"b" "a"
d	"a-b"
e	"d" "a-b"
+	"e" "d" "a-b"
f	"d+e" "a-b"
*****	"f" "d+e" "a-b"
/	"d+e"*f "a-b"
	"a-b"/"d+e"*f
NULL	**"((a-b)/((d-e)*f))"**

So, the Infix Expression is
((a-b)/((d-e)*f))

3. Postfix to Infix

Example: Convert Postfix to Infix efg-+he-sh-o+/*

efg-+he-sh-o+/*	NULL
fg-+he-sh-o+/*	"e"
g-+he-sh-o+/*	"f" "e"
-+he-sh-o+/*	"g" "f" "e"
+he-sh-o+/*	"f"-"g" "e"
he-sh-o+/*	"e+f-g"
e-sh-o+/*	"h" "e+f-g"
-sh-o+/*	"e" "h" "e+f-g"
sh-o+/*	"h-e" "e+f-g"
h-o+/*	"s" "h-e" "e+f-g"
-o+/*	"h" "s" "h-e" "e+f-g"
o+/*	"h-s" "h-e" "e+f-g"
+/*	"o" "s-h" "h-e" "e+f-g"
/*	"s-h+o" "h-e" "e+f-g"
*	"(h-e)/(s-h+o)" "e+f-g"
NULL	**"(e+f-g)* (h-e)/(s-h+o)"**

So, the Infix Expression is
(e+f-g)* (h-e)/(s-h+o)

DACC

Que 1. Solve the following expression
5,6,2+,*,12,4,/,-

Step	Symbol	Operator in Stack
1	5	5
2	6	5,6
3	2	5,6,2
4	+	5,8
5	*	40
6	12	40,12
7	4	40,12,4
8	/	40,3
9	-	37

Answer = 37

Que 2. Solve the following expression
15,5,10,+,3,/,+

Step	Symbol	Operator in Stack
1	15	15
2	5	15,5
3	10	15,5,10
4	+	15,15
5	3	15,15,3
6	/	15,5
9	+	20

Answer = 20

Example : 4,5,4,2,^,+,*,2,2,^,7,3,/,*,-

Step	Symbol	Operator in Stack
1	4	4
2	5	4,5
3	4	4.5.4
4	2	4,5,4,2
5	^	4,5,16
6	+	4,21
7	*	84
8	2	84,2
9	2	84,2,2
10	^	84,4
11	9	84,4,9
12	3	84,4,9,3
13	/	84,4,3
14	*	84,12
15	-	72

Answer = 72

UNT 5. Queues

4.1 Introduction and Definition

4.2 Representation of Queues

4.3 Operation on Queues

4.4 Applications of Queues

4.5 Dequeue

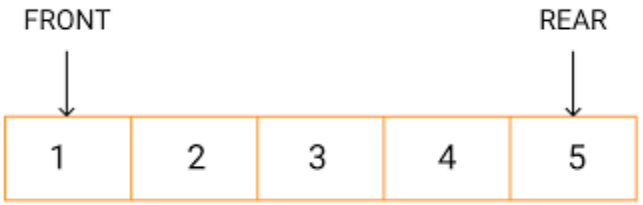
4.6 Circular Queue

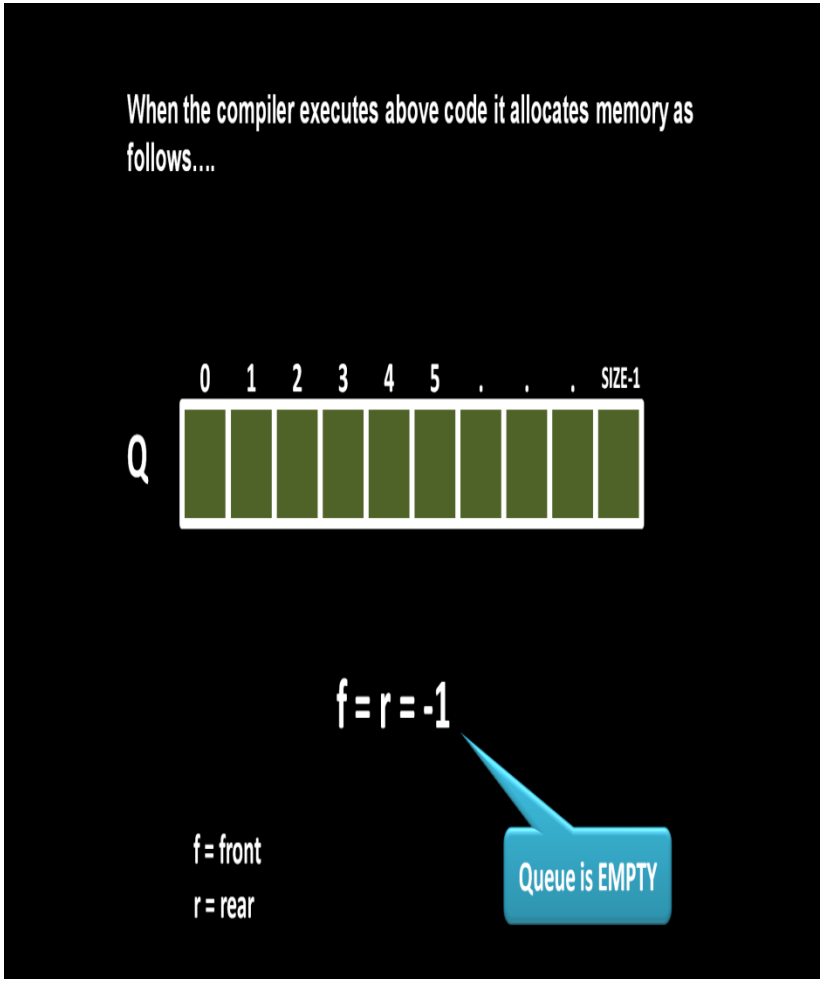
4.7 Priority Queue

Unit 2. Queue

What is Queue?

- Queue is a linear data structure where the first element is inserted from one end called **REAR** and deleted from the other end called as **FRONT**.
- **Front** points to the **beginning** of the queue and **Rear** points to the **end** of the queue.
- Queue follows the **FIFO (First - In - First Out)** structure.
- According to its FIFO structure, element inserted first will also be removed first.
- In a queue, one end is always used to insert data and the other is used to delete data.



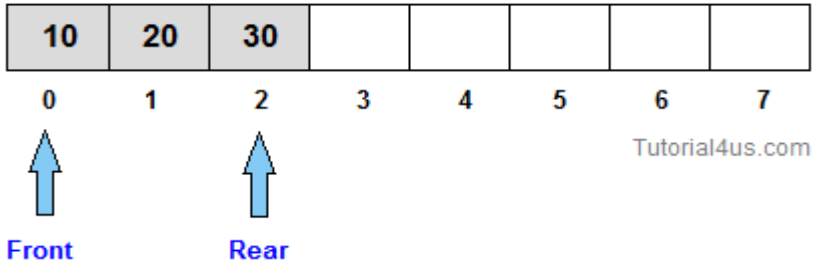


Operation on a queue

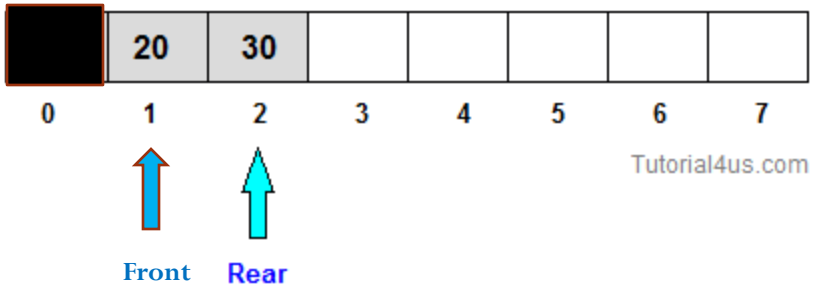
The basic operation that can be perform on queue are;

1. Insert an Element in a Queue.
2. Delete an Element from the Queue.

1. Insert an element in a queue.



2. Delete an Element from the Queue.



Implementation of Queue

- **Array** is the easiest way to implement a queue. Queue can be also implemented using Linked List or Stack.

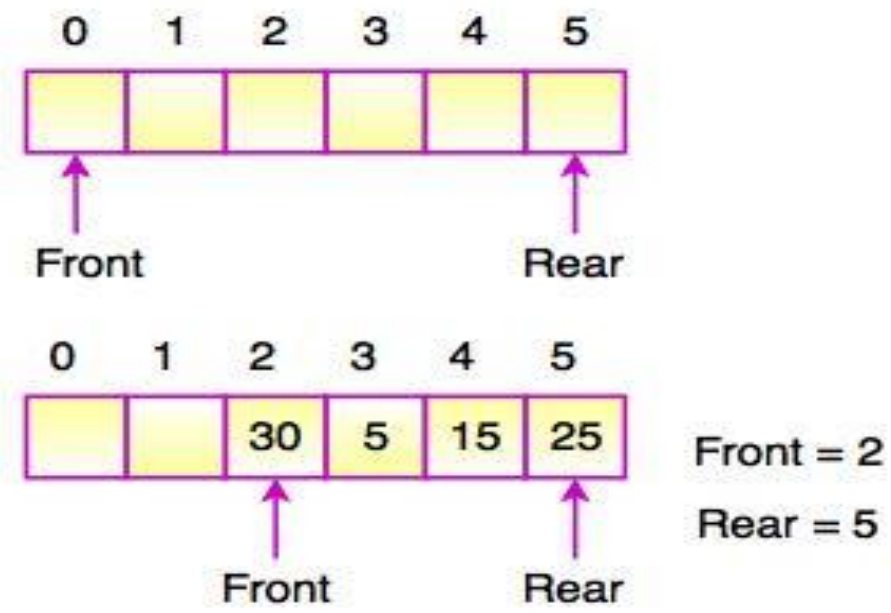


Fig. Implementation of Queue using Array

In the above diagram, Front and Rear of the queue point at the first index of the array. (Array index starts from 0).

While adding an element into the queue, the Rear keeps on moving ahead and always points to the position where the next element will be inserted. Front remains at the first index.

Queue Applications

Real-World Applications

- Buy a movie ticket
- Check out at a bookstore
- Bank / ATM
- Call an airline
- Cashier lines in any store

Computer Science Applications

- OS task scheduling
- Print lines of a document
- Printer shared between computers
- Convert digit strings to decimal
- Shared resource usage (CPU, memory access, **...**)

Functions of Queue

insert()	delete()	display()

insert()
{
int add_item;
if (rear == MAX - 1)
{
printf("Queue Overflow \n");
}
else
{
if (front == - 1)
{
front = 0;
}
printf("Inset the element in queue : ");
scanf("%d", &add_item);
rear = rear + 1;
queue_array[rear] = add_item;
}
}
	delete()
	{
	if (front == - 1 || front > rear)
	{
	printf("Queue Underflow \n");
	}
	else
	{
	printf("Deleted Element is : %d\n", queue_array[front]);
	front = front + 1;
	}
	}
		display()
		{
		int i;
		if (front == - 1)
		{
		printf("Queue is empty \n");
		}
		else
		{
		for (i = front; i <= rear; i++)
		{
		printf("%d \n", queue_array[i]);
		}
		}
		}

DACC

Program of Queue using Array

```
#include <stdio.h>
#define MAX 50
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
   int choice;
   while (1)
   {
      printf("1.Insert \n");
      printf("2.Delete\n");
      printf("3.Display \n");
      printf("4.Exit \n");
      printf("Enter your choice : ");
      scanf("%d", &choice);
      switch (choice)
      {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(1);
         default:
         printf("Inavlid choice \n");
      }
   }
}
```

```
insert()
{
   int add_item;
   if (rear == MAX - 1)
   {
   printf("Queue Overflow \n");
   }
 else
   {
      if (front == - 1)
         {
            front = 0;
         }
      printf("Inset the element in queue : ");
      scanf("%d", &add_item);
      rear = rear + 1;
      queue_array[rear] = add_item;
   }
}
delete()
{
   if (front == - 1 || front > rear)
   {
      printf("Queue Underflow \n");
    }
   else
   {
      printf("Deleted Element is : %d\n",
queue_array[front]);
      front = front + 1;
   }
}
```

```
display()
{
   int i;
   if (front == - 1)
    {
      printf("Queue is empty \n");
    }
   else
   {
       for (i = front; i <= rear; i++)
            {
              printf("%d ", queue_array[i]);
      }}}
```

1. Insert Element in Queue

```
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
Inset the element in queue : 10
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
Inset the element in queue : 20
```

2. Display Element

```
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 3
Queue is :
10 20 30 40
```

3. Delete Element

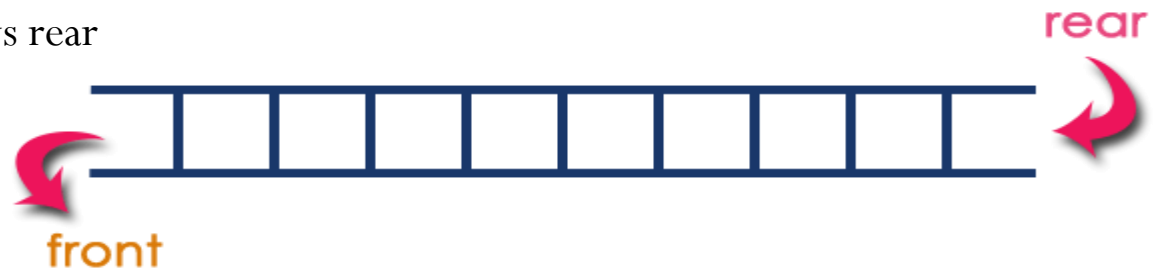
```
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 2
Deleted Element is : 10
```

Types of Queue in Data Structure

- **There are four types of Queue:**
  1. Linear Queue
  2. Circular Queue
  3. Priority Queue
  4. Dequeue (Double Ended Queue)1. Simple Queue

1. Linear Queue

In this queue the elements are arranged into sequential manner. Such that front position is always less than or equal to the rear position. When an element is added, rear is incremented and when an element is removed front is advanced. Thus front always rear

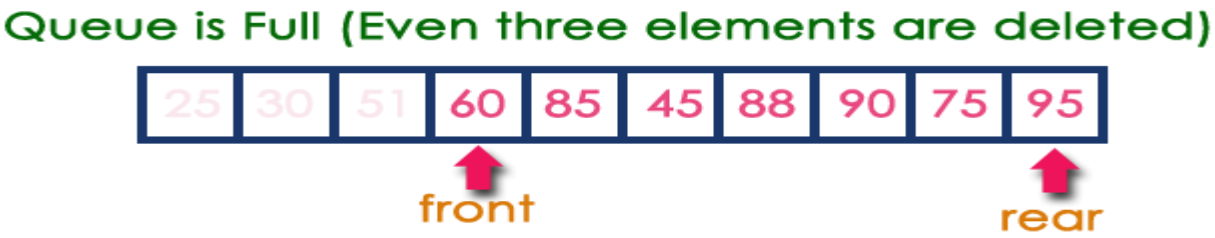


2. Circular Queue

In this queue , the elements are arranged in a sequential manner but can logically be regarded as
circularly arranged.

- In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we can not insert the next element until all the elements are deleted from the queue.
- For example, consider the queue below…The queue after inserting all the elements into it is as follows…

Now consider the following situation after deleting three elements from the queue…



This situation also says that Queue is Full and we cannot insert the new element because '**rear**' is still at last position. In the above situation, even though we have empty positions in the queue we can not make use of them to insert the new element. This is the major problem in a normal queue data structure. To overcome this problem we use a circular queue data structure.

In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.

- Circular queue is also called as **Ring Buffer.**
- It is an abstract data type.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

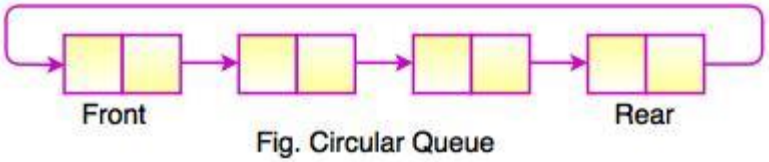


Fig. Circular Queue

The above figure shows the structure of circular queue. It stores an element in a circular way and performs the operations according to its FIFO structure.

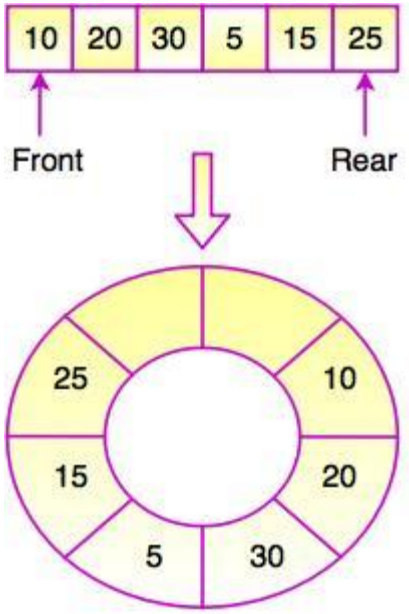
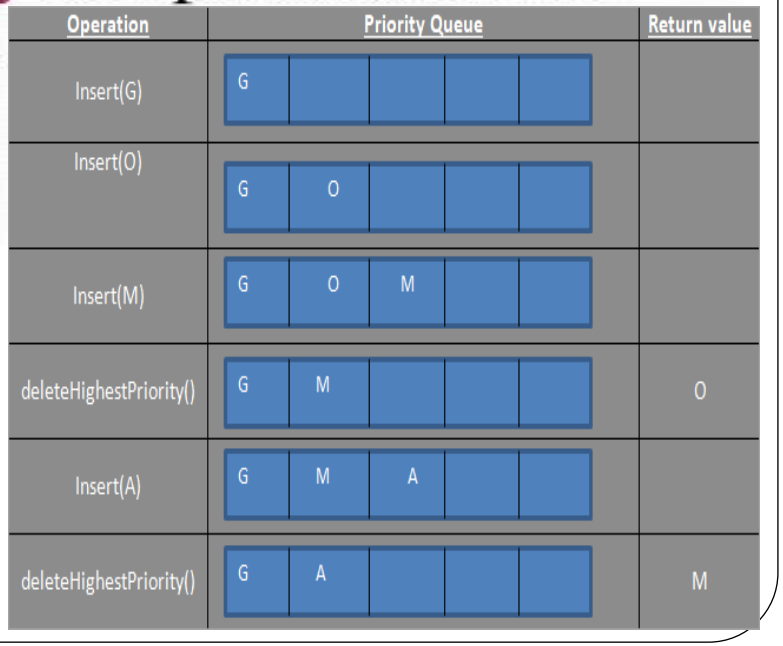


Fig. Circular Queue

PRIORITY QUEUE

1. It is collection of elements where elements are stored according to the their priority levels.
2. Inserting and removing of elements from queue is decided by the priority of the elements.
3. An element of the higher priority is processed first.
4. Two element of same priority are processed on first-come-first-served basis.



Applications of Priority Queue:

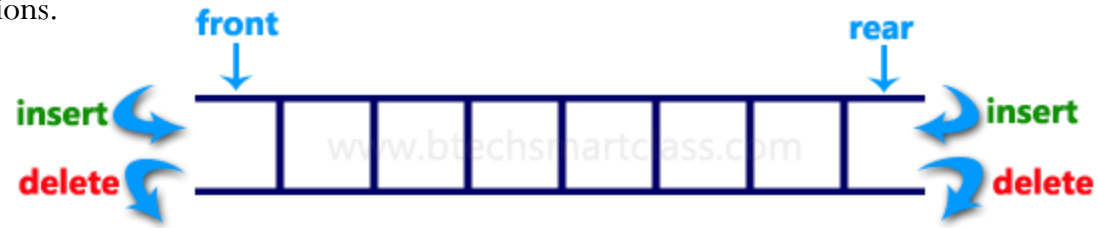
1) CPU Scheduling

2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc

3) All queue applications where priority is involved.

4. Dequeue (Double Ended Queue)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

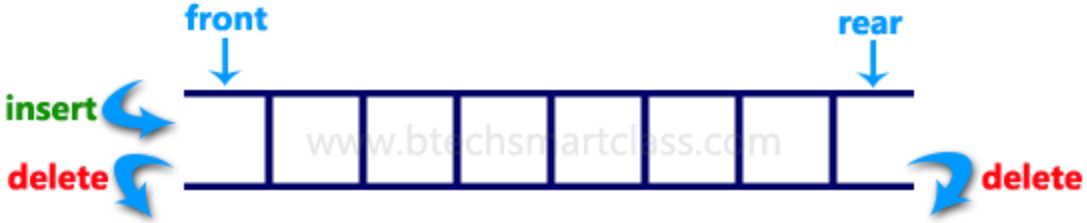


Double Ended Queue can be represented in TWO ways, those are as follows…

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

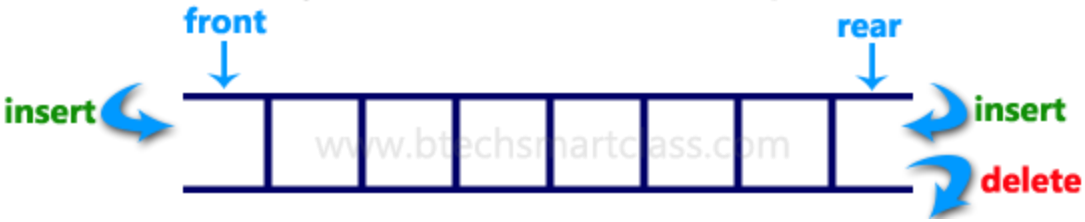
1. Input Restricted Double Ended Queue

In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



2. Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



UNIT 6 : Trees

6.1 Definition of Tree

6.2 Binary Tree and their types

6.3 Representation of Binary Tree

6.4 Operations on Binary Tree

6.5 Binary Search Tree (BST)

6.6 Traversal of Binary Tree

6.6.1 Pre-order Traversal

6.6.2 In-order Traversal

6.6.3 Post-order Traversal

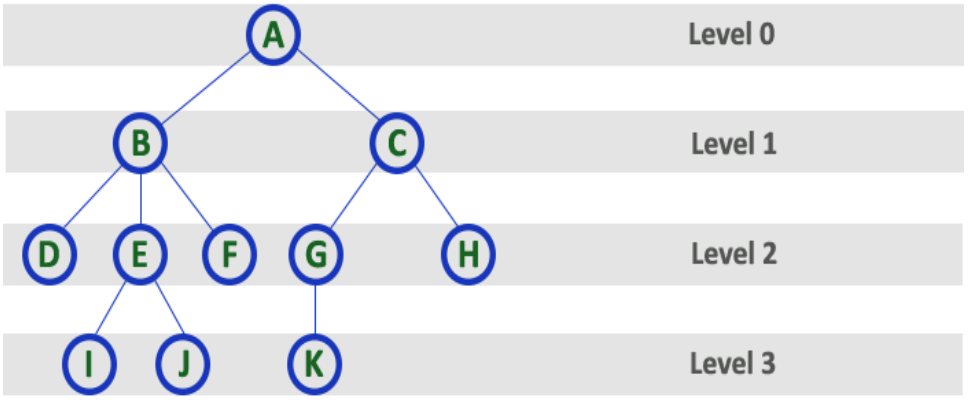
6.7 Threaded Binary Tree

6.8 AVL Tree

What is Tree?

- Tree is a non linear data structure which represents hierarchical relationship among its elements.
- Along with information storage tree as a data structure is efficient with respect to operations such as Insertion, Deletion, Searching as compared to linear data structure.
- **Properties of Tree :**

1. There exists unique path between every two vertices.
2. The number of vertices is one more than the no of edges in tree
3. A tree with two or more vertices has at least two leaves.



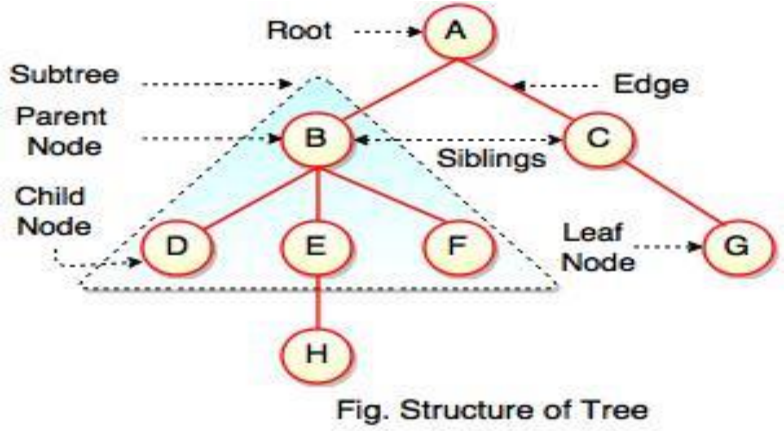
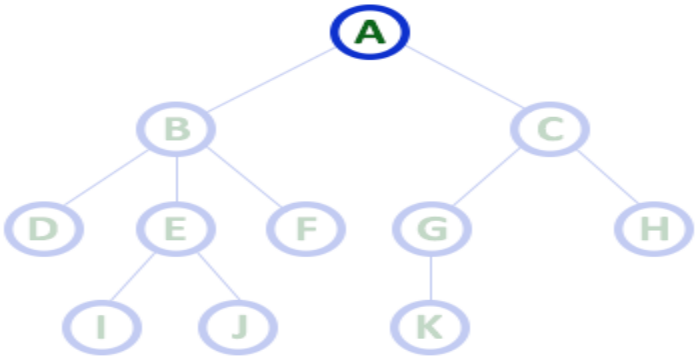


Fig. Structure of Tree

Tree - Terminology

1. Root

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

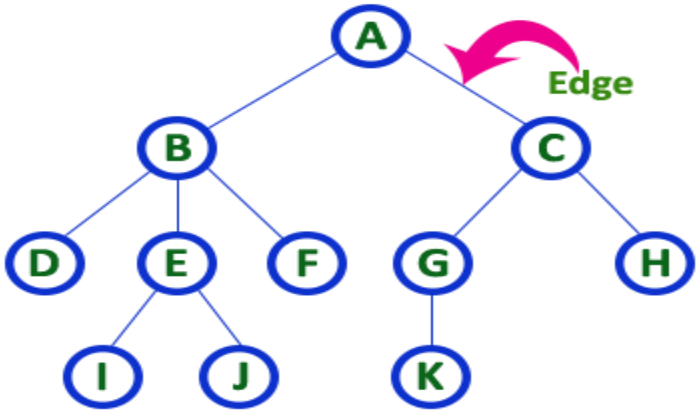


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

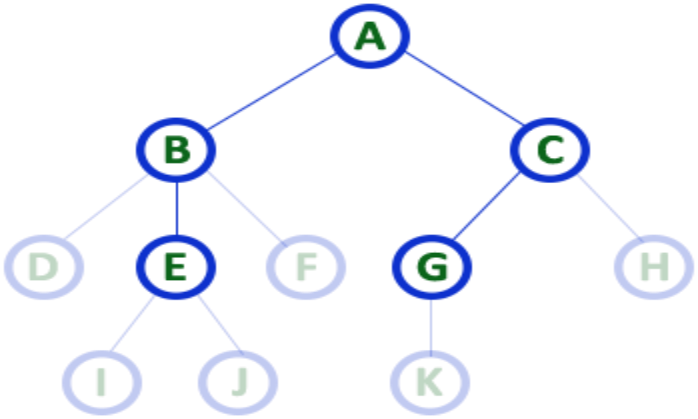
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

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

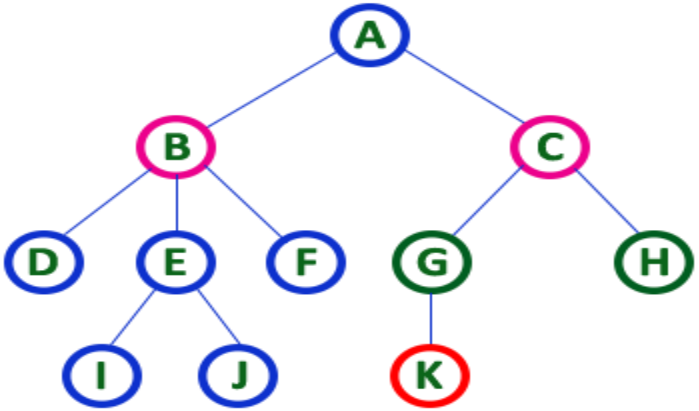


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child

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

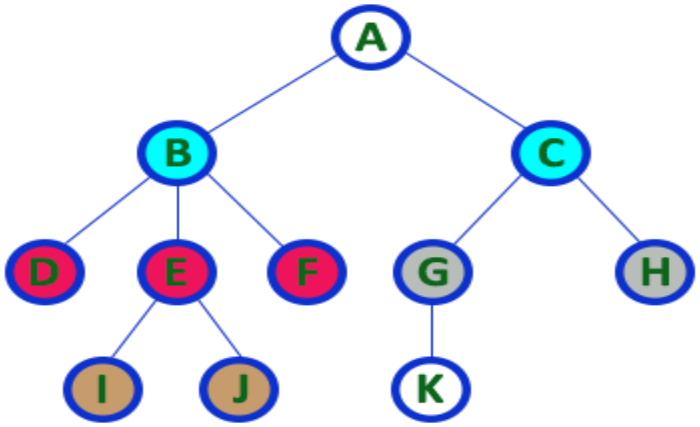


Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

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



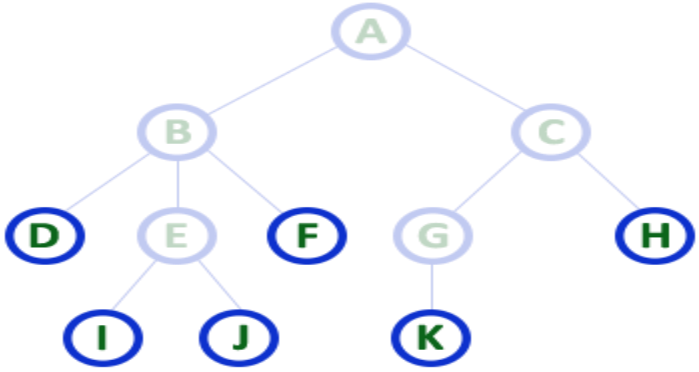
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



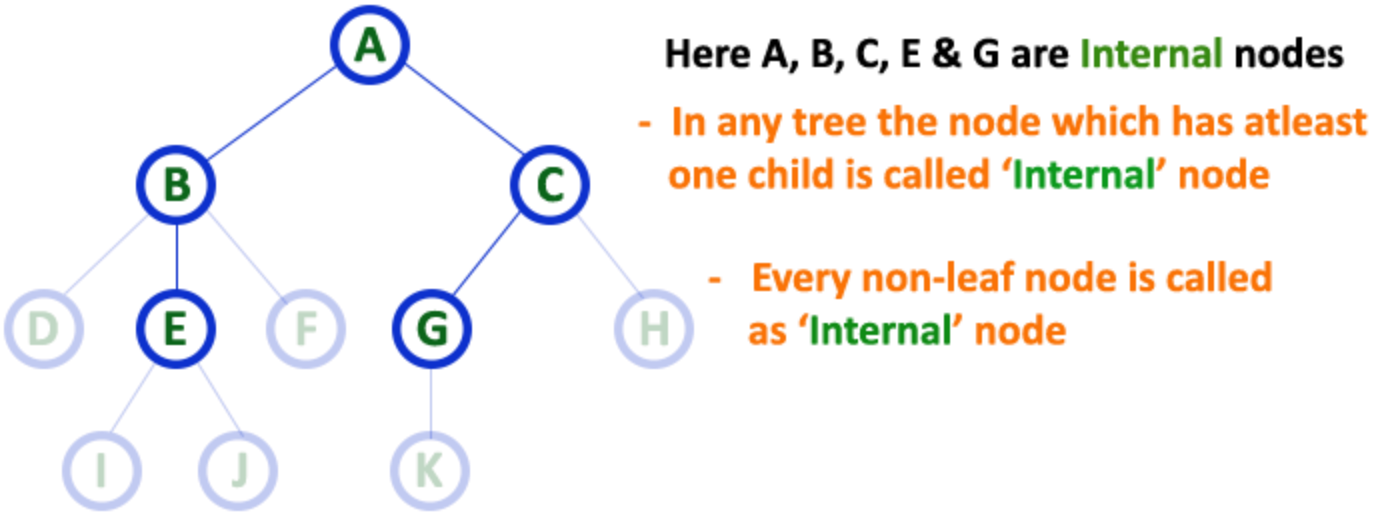
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

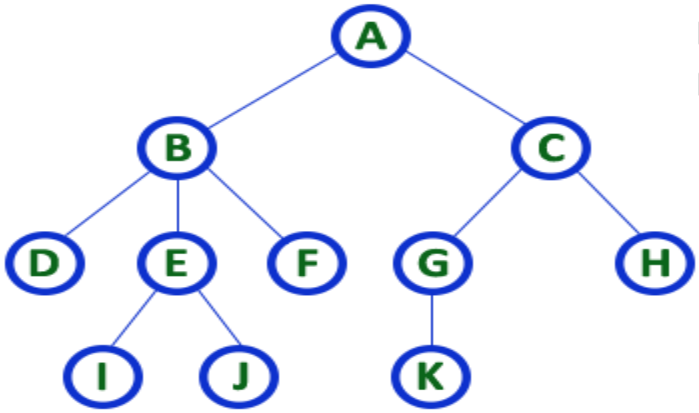
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



8. Degree

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



Here Degree of B is 3

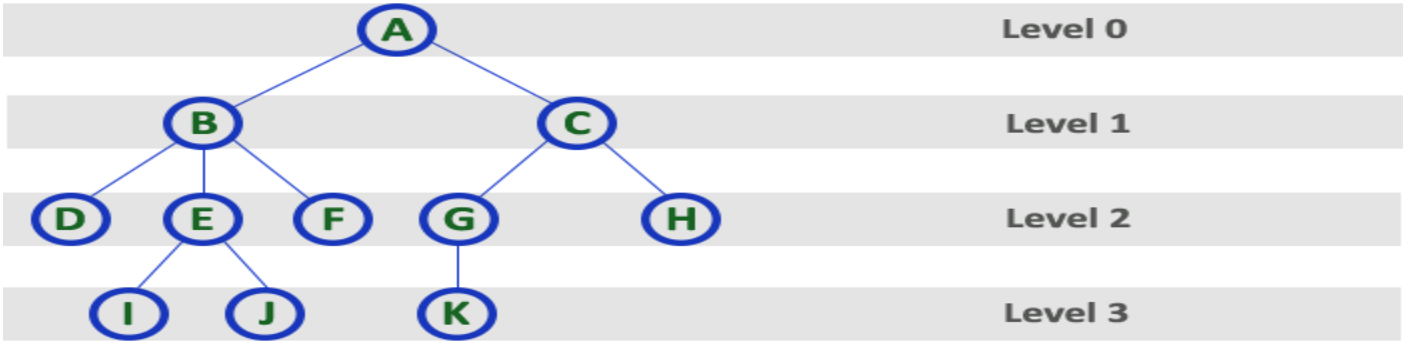
Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

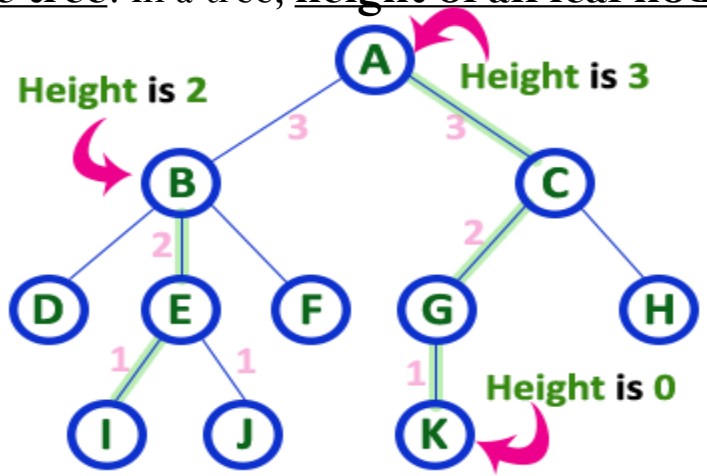
9. Level

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



10. Height

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

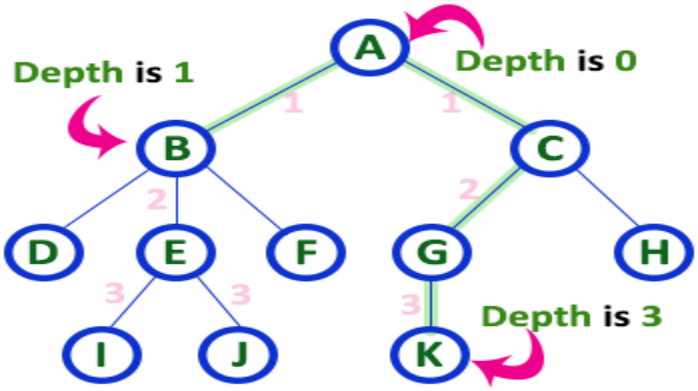


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

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

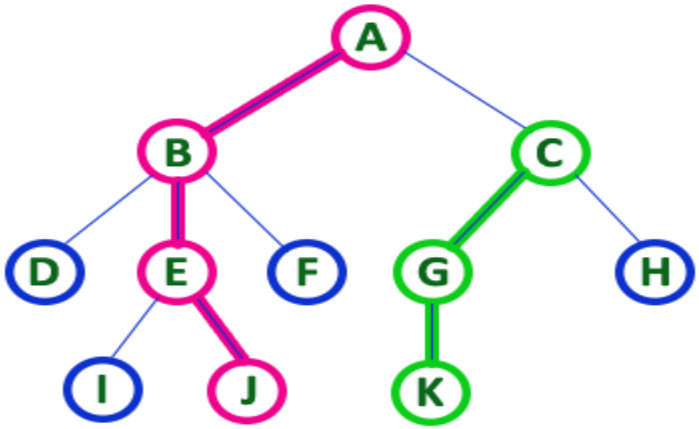


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

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



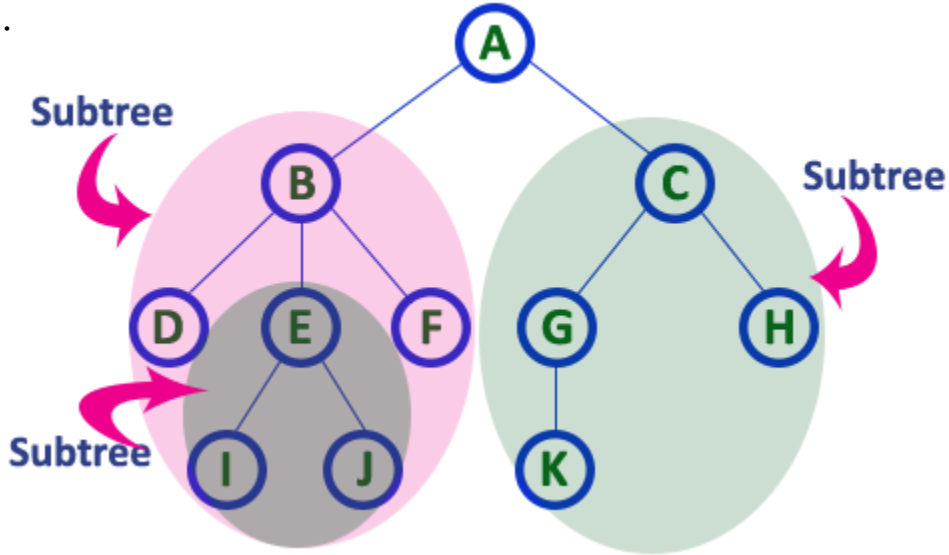
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

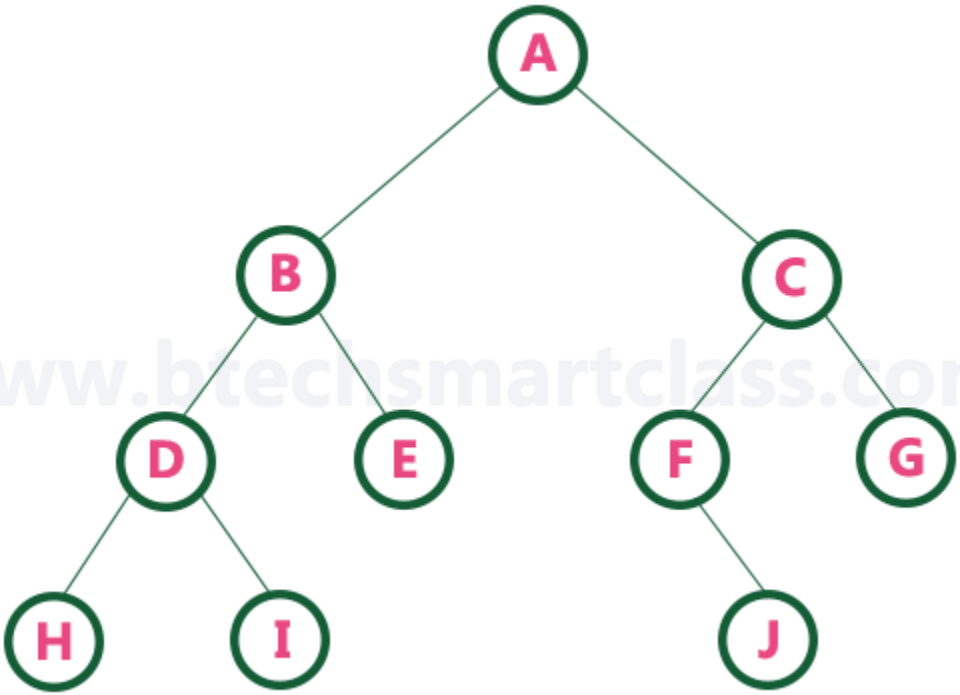
13. Sub Tree

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



Binary Tree and their types

- In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as a left child and the other is known as right child.
- **A tree in which every node can have a maximum of two children is called Binary Tree.**
- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children



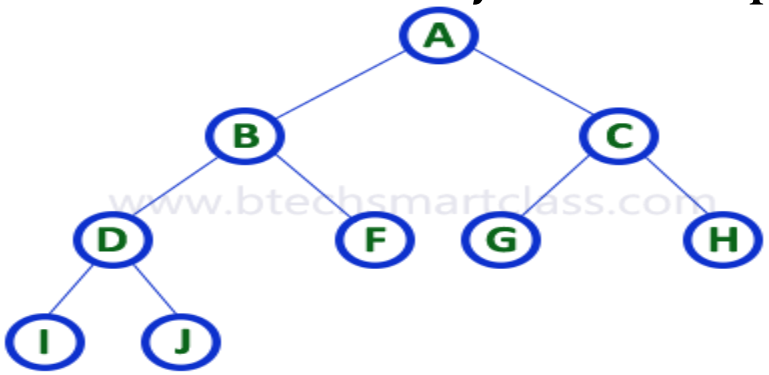
Type of Binary Tree

There are different types of binary trees and they are…

1. Strictly Binary Tree

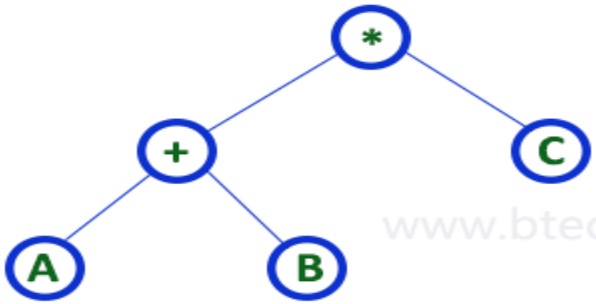
In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows…

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

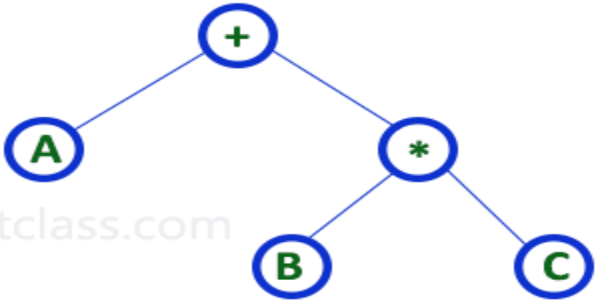


Strictly binary tree data structure is used to represent mathematical expressions.

Example



(A + B) * C



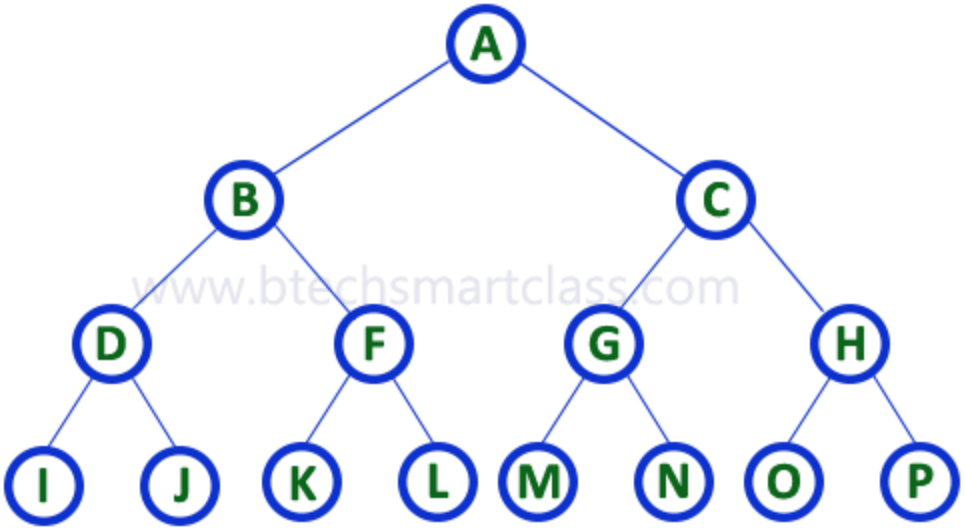
A + B * C

2. Complete Binary Tree

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

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

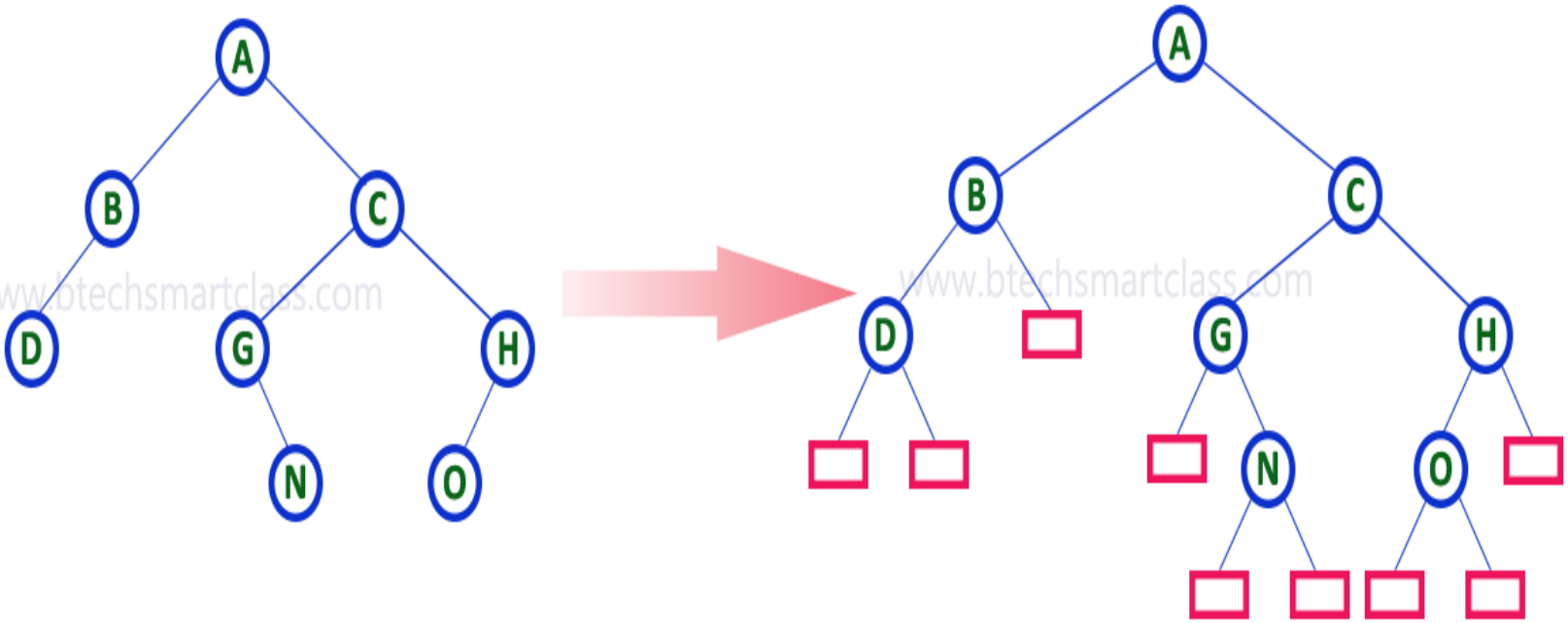
Complete binary tree is also called as **Perfect Binary Tree**



3. Extended Binary Tree

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

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



4 Skewed Binary Tree

- If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.
- In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.

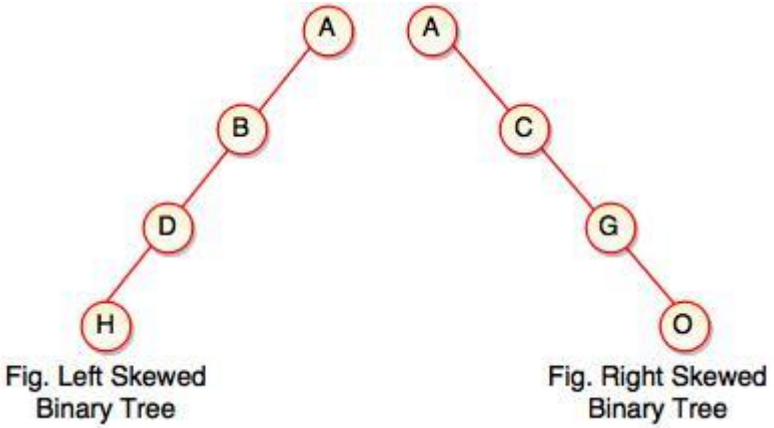


Fig. Left Skewed Binary Tree

Fig. Right Skewed Binary Tree

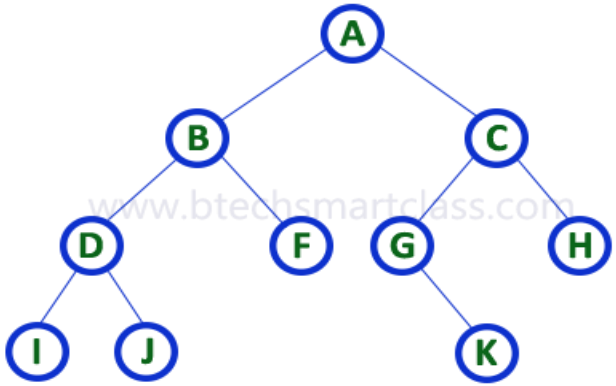
- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.

Binary Tree Representations

- A binary tree data structure is represented using two methods. Those methods are as follows…

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree…



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows…

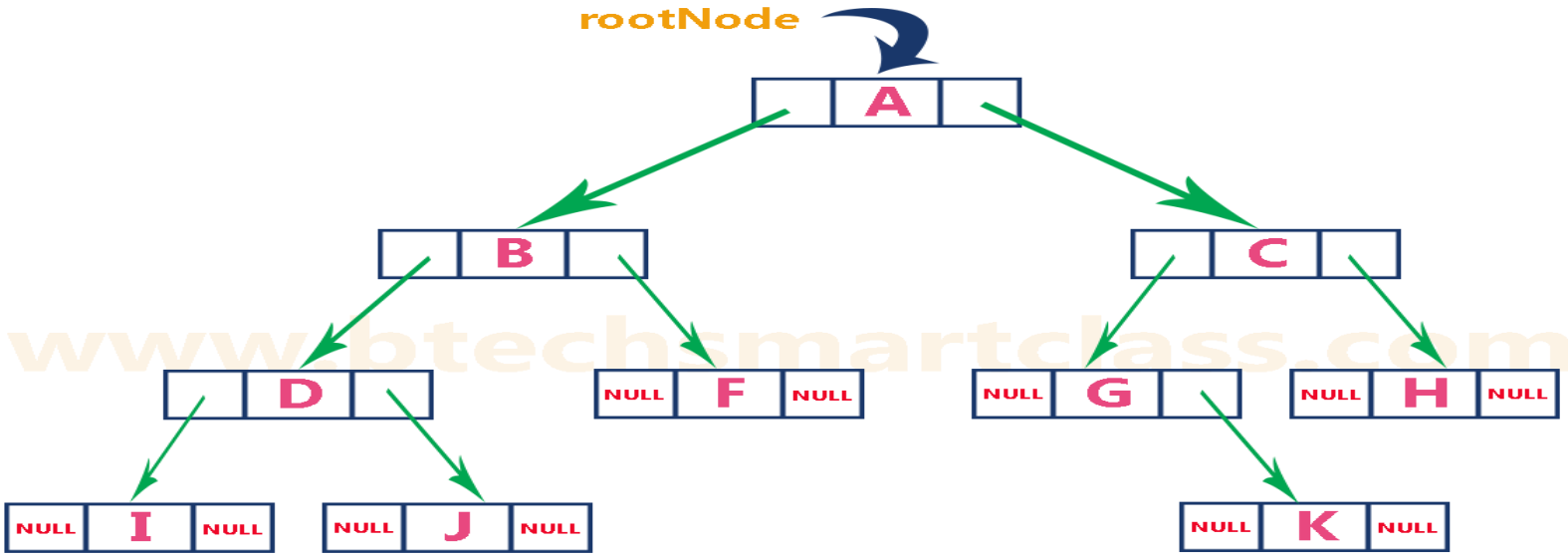
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure…

The above example of the binary tree represented using Linked list representation is shown as follows..



Binary Search Tree

- Binary search tree is a binary tree which has special property called BST.
- BST property is given as follows:
- **For all nodes A and B,**

  I. If B belongs to the left subtree of A, the key at B is less than the key at A.

  II. If B belongs to the right subtree of A, the key at B is greater than the key at A.

  Each node has following attributes:

  I. Parent (P), left, right which are pointers to the parent (P), left child and right child respectively.

  II. Key defines a key which is stored at the node.

Definition:

"Binary Search Tree is a binary tree where each node contains only smaller values in its left subtree and only larger values in its right subtree."

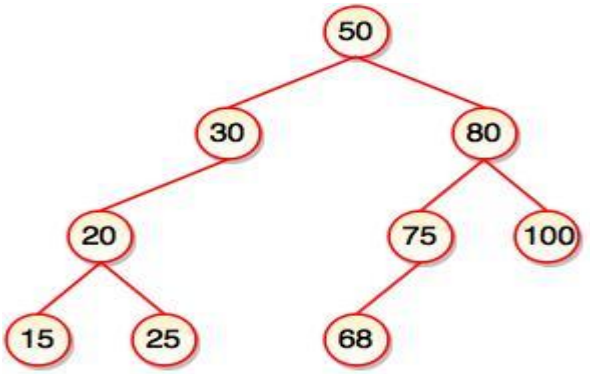
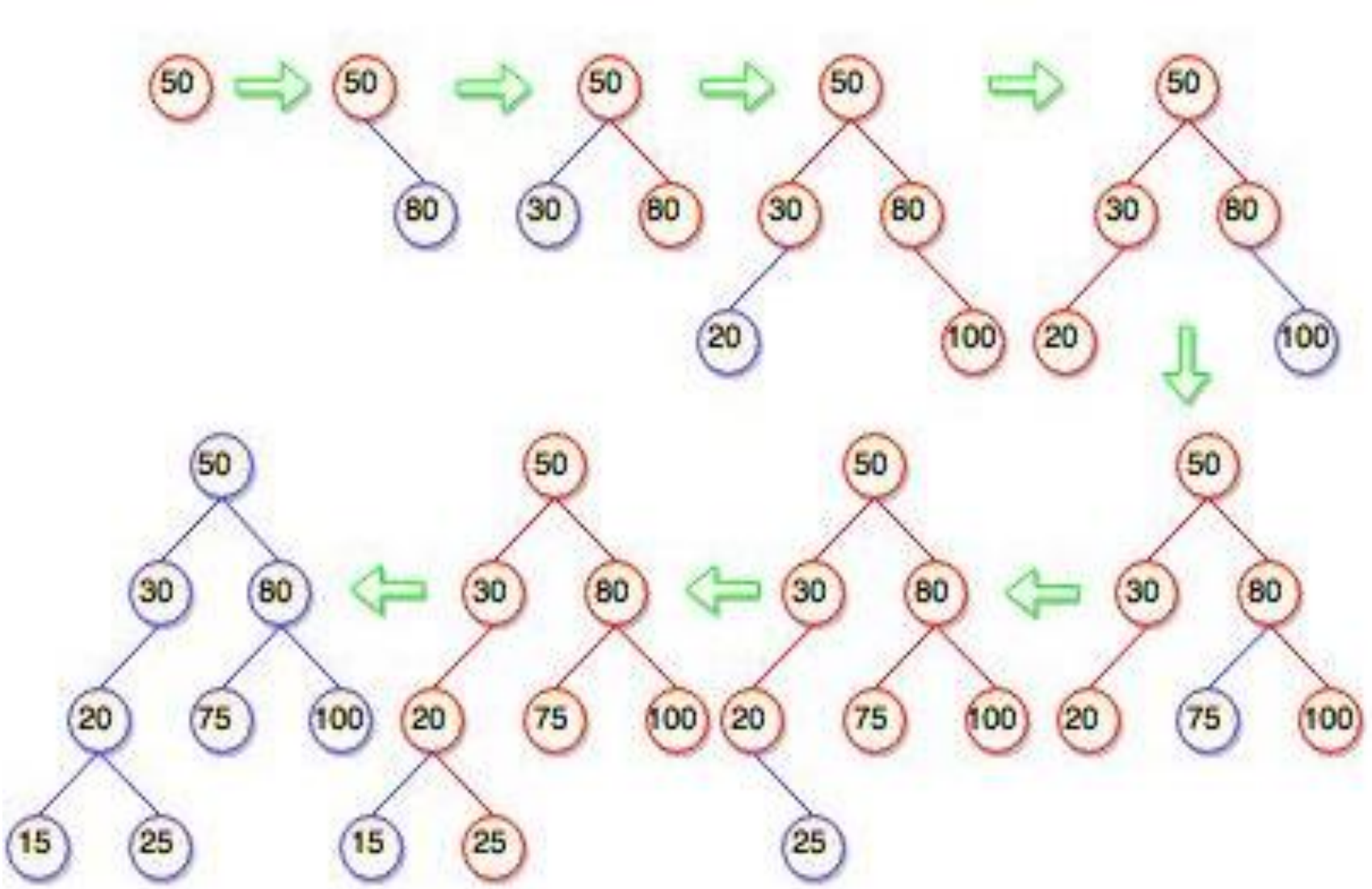


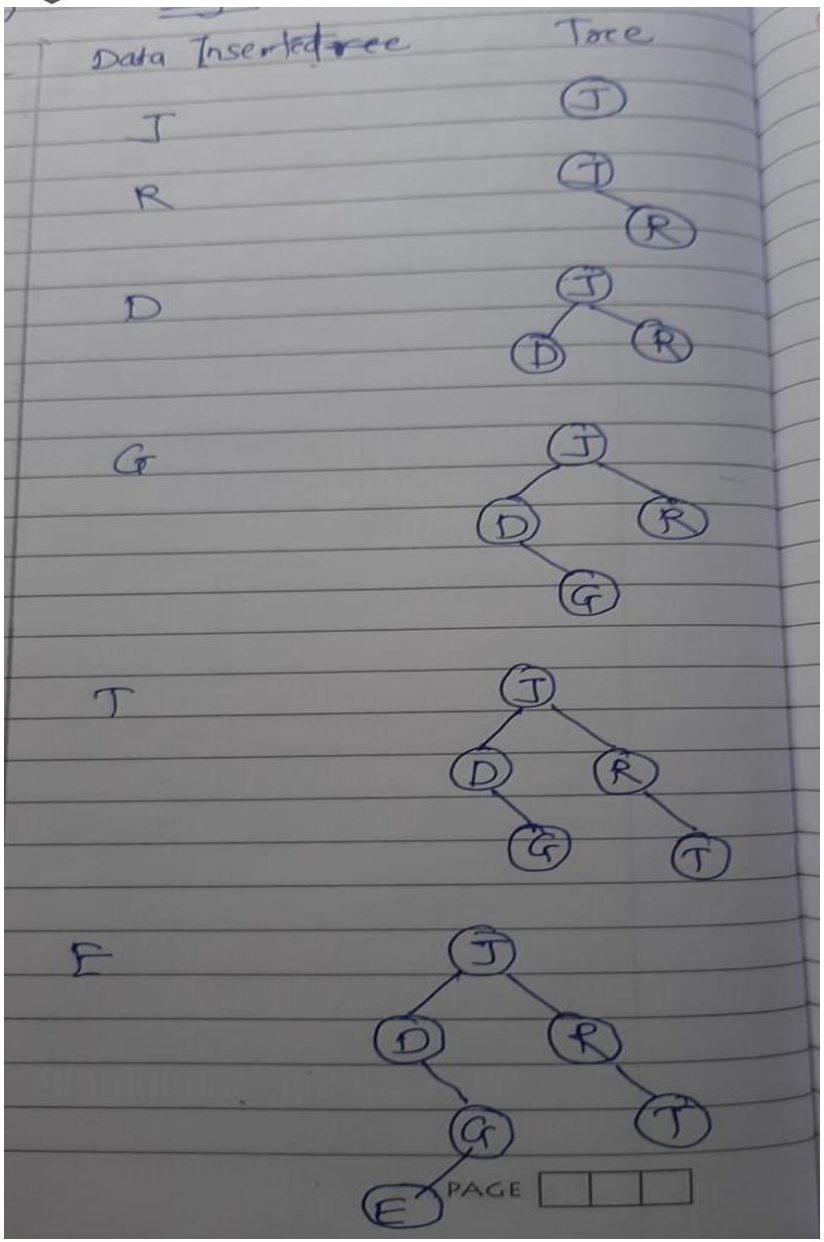
Fig. Binary Search Tree

- The tree represents binary search tree (BST) where left subtree of every node contains smaller values and right subtree of every node contains larger value.
- Binary Search Tree (BST) is used to enhance the performance of binary tree.
- It focuses on the search operation in binary tree.

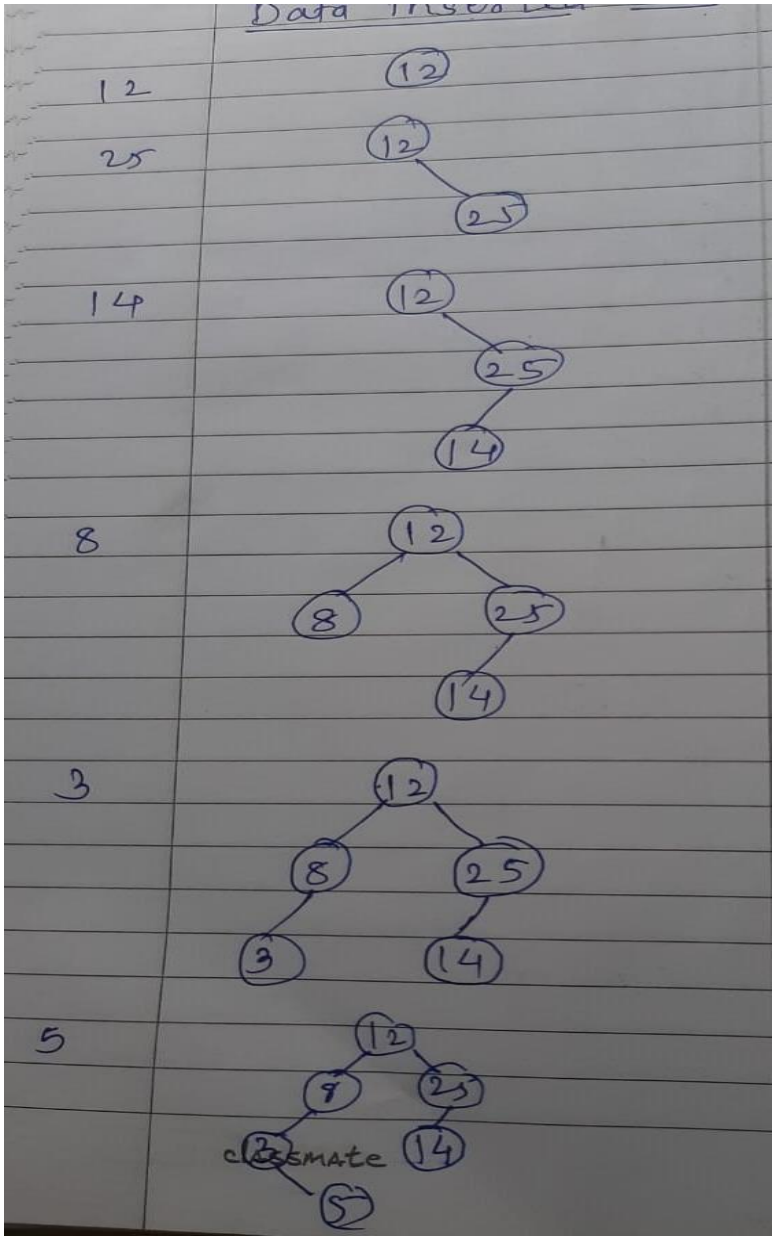
Create a Binary Search tree
50,80,30,20,100,75,25,15



Create a Binary Search tree
J,R,DG,T,E



Create a Binary Search tree
12,25,14,8,3,5



Binary Tree Traversal

- Binary tree traversing is a process of accessing every node of the tree and exactly once. A tree is defined in a recursive manner. Binary tree traversal also defined recursively.

There are three techniques of traversal:

1. Preorder Traversal **(NLR)**
2. Postorder Traversal **(LRN)**
3. Inorder Traversal **(LNR)**

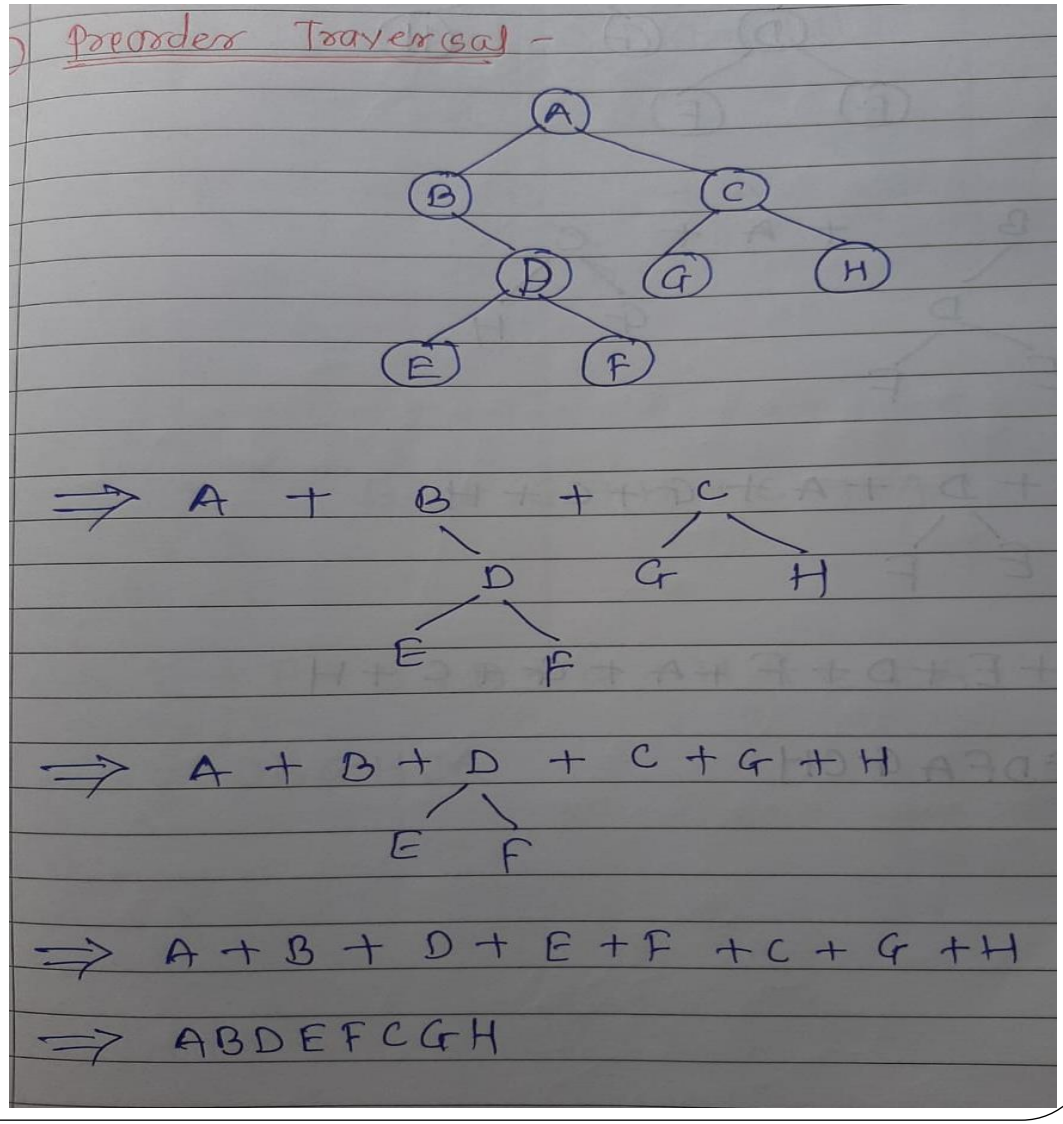
1. Preorder Traversal(NLR)

- **Algorithm for preorder traversal**

  Step 1 : Start from the Root.

  Step 2 : Then, go to the Left Subtree.

  Step 3 : Then, go to the Right Subtree.



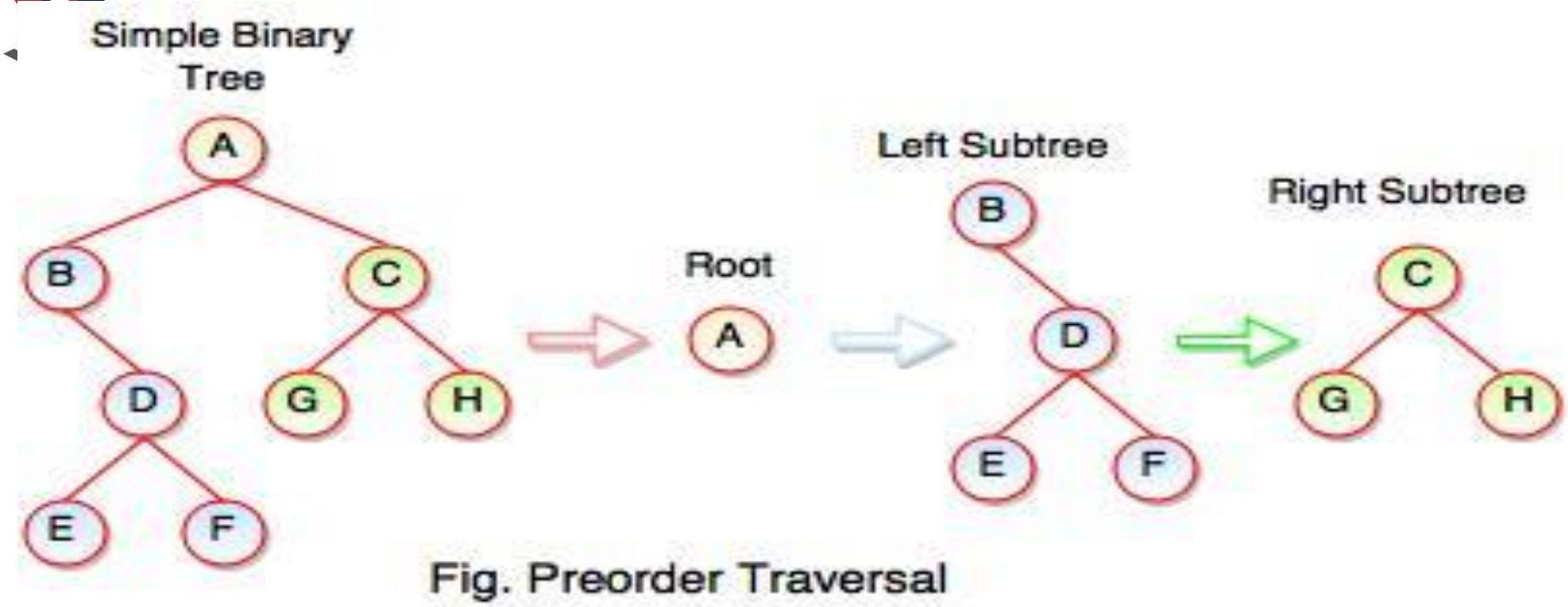


Fig. Preorder Traversal

Step 1 : A + B (B + Preorder on D (D + Preorder on E and F)) + C (C + Preorder on G and H)

Step 2 : A + B + D (E + F) + C (G + H)

Step 3 : A + B + D + E + F + C + G + H

Preorder Traversal : A B C D E F G H

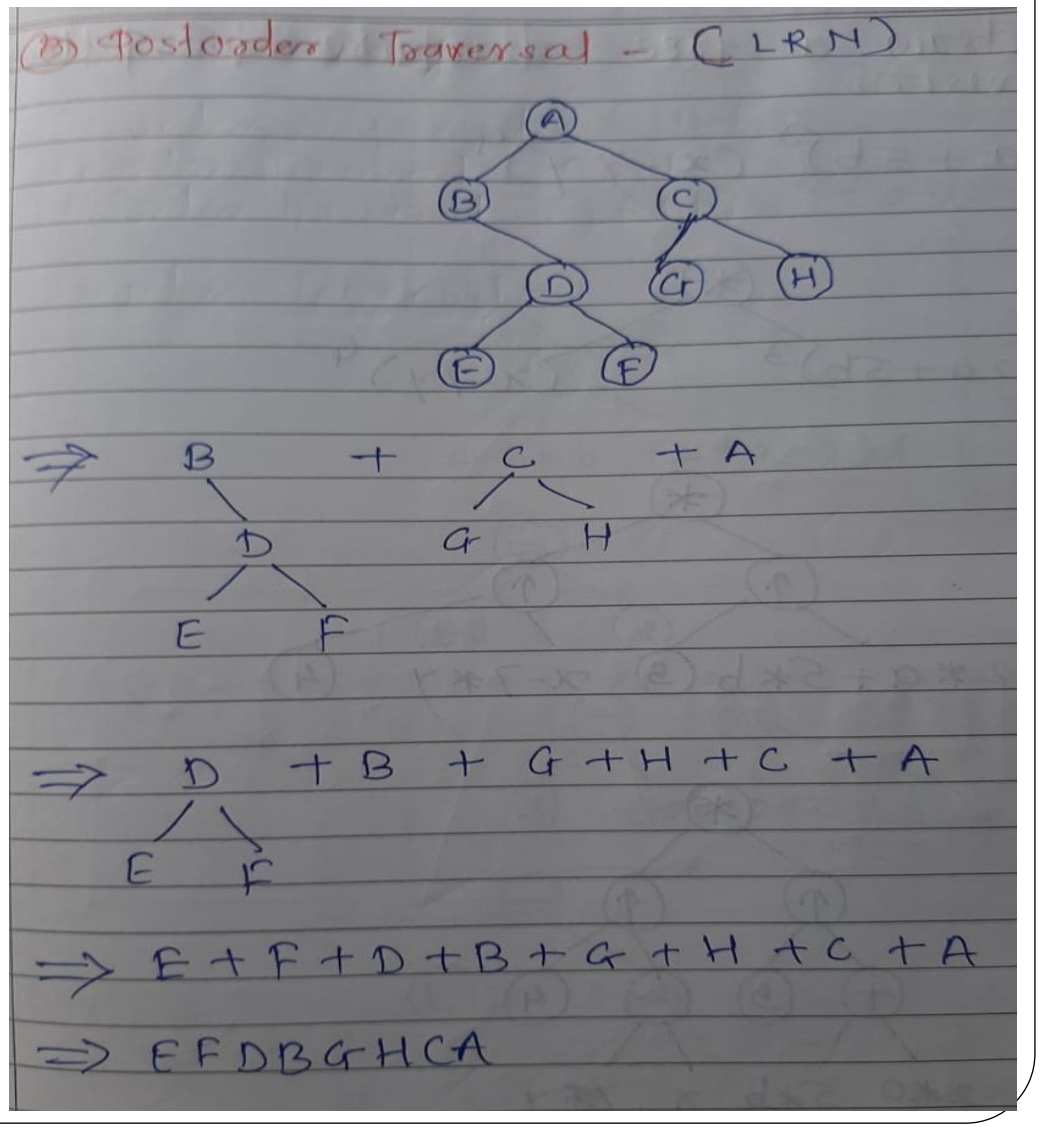
Postorder Traversal(LRN)

Algorithm for postorder traversal :

Step 1 : Start from the Left Subtree (Last Leaf).

Step 2 : Then, go to the Right Subtree.

Step 3 : Then, go to the Root.



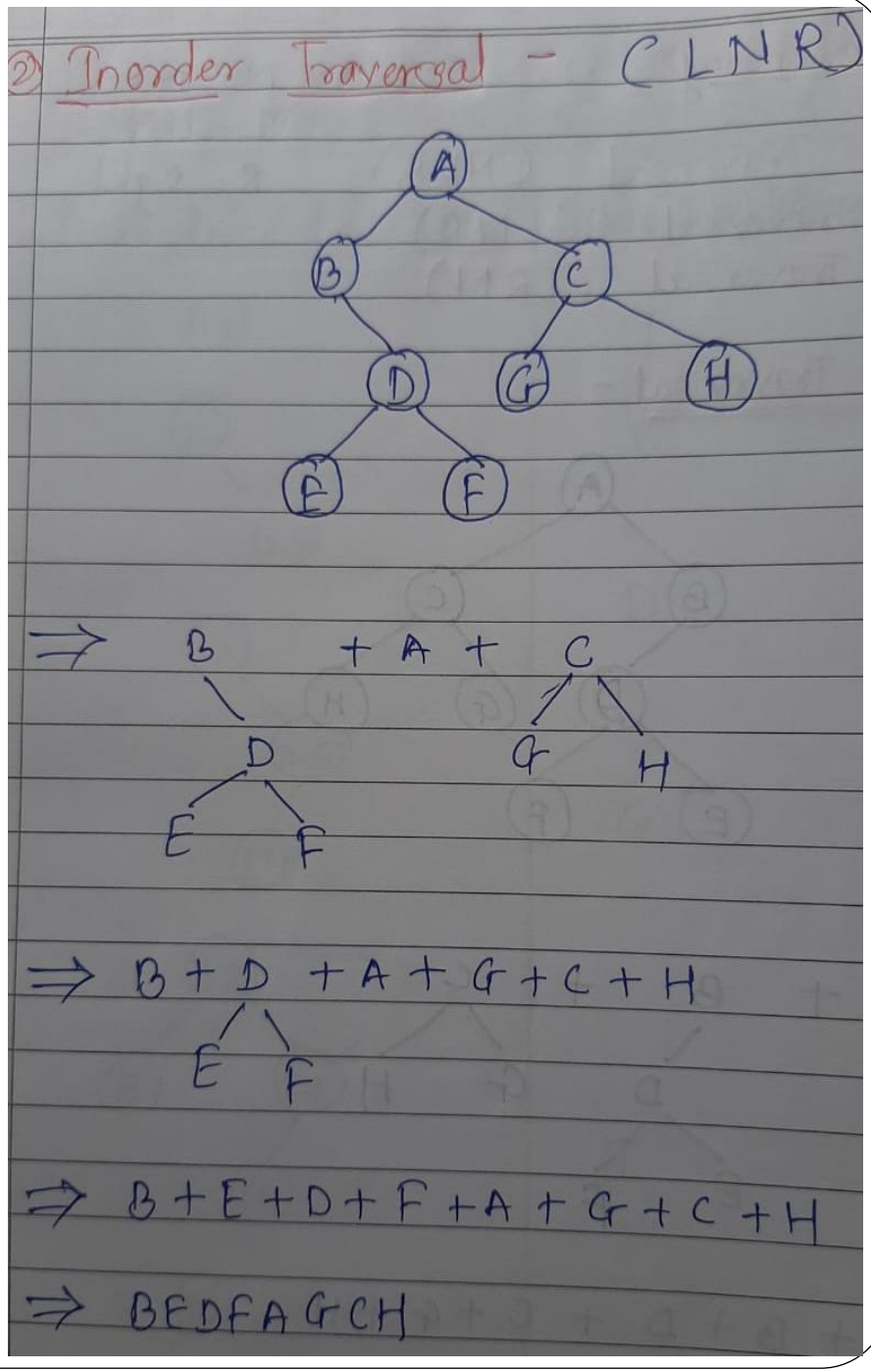
3. Inorder Traversal

Algorithm for inorder traversal

Step 1 : Start from the Left Subtree.

Step 2 : Then, visit the Root.

Step 3 : Then, go to the Right Subtree.



AVL Tree

AVL-Tree (Balanced Tree)

DATE

Addson, Velski & Landis (AVL) in 1962 introduced a tree structure that is balanced with respect to the height of the subtrees. such a tree is called as AVL Tree.

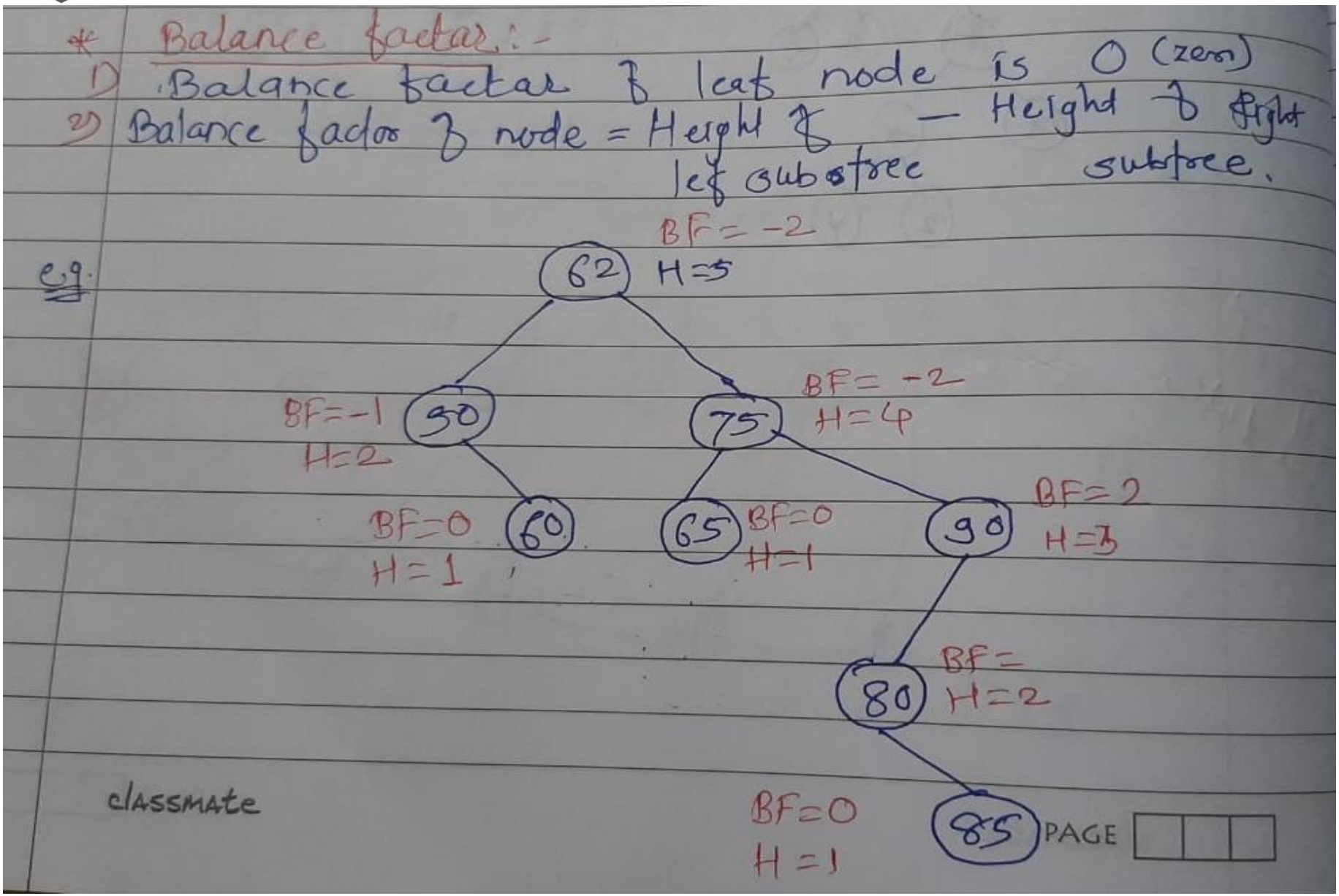
An AVL Tree is binary search tree where height of left and right subtree of any node will be with maximum difference 1.

* **Height of Node :-**

1) Height of leaf node is always 1

2) Height of Internal node = 1 + max [left child, Right child (Height)]

Balance Factor



Height of Node

1) Height of leaf node is 1

∴ Height(85)
Height(65) = } = 1
Height(60)

2) Height of (80) = 1 + max [left child, Height(85)]
= 1 + max [-, 1]
= 1 + 1
= 2

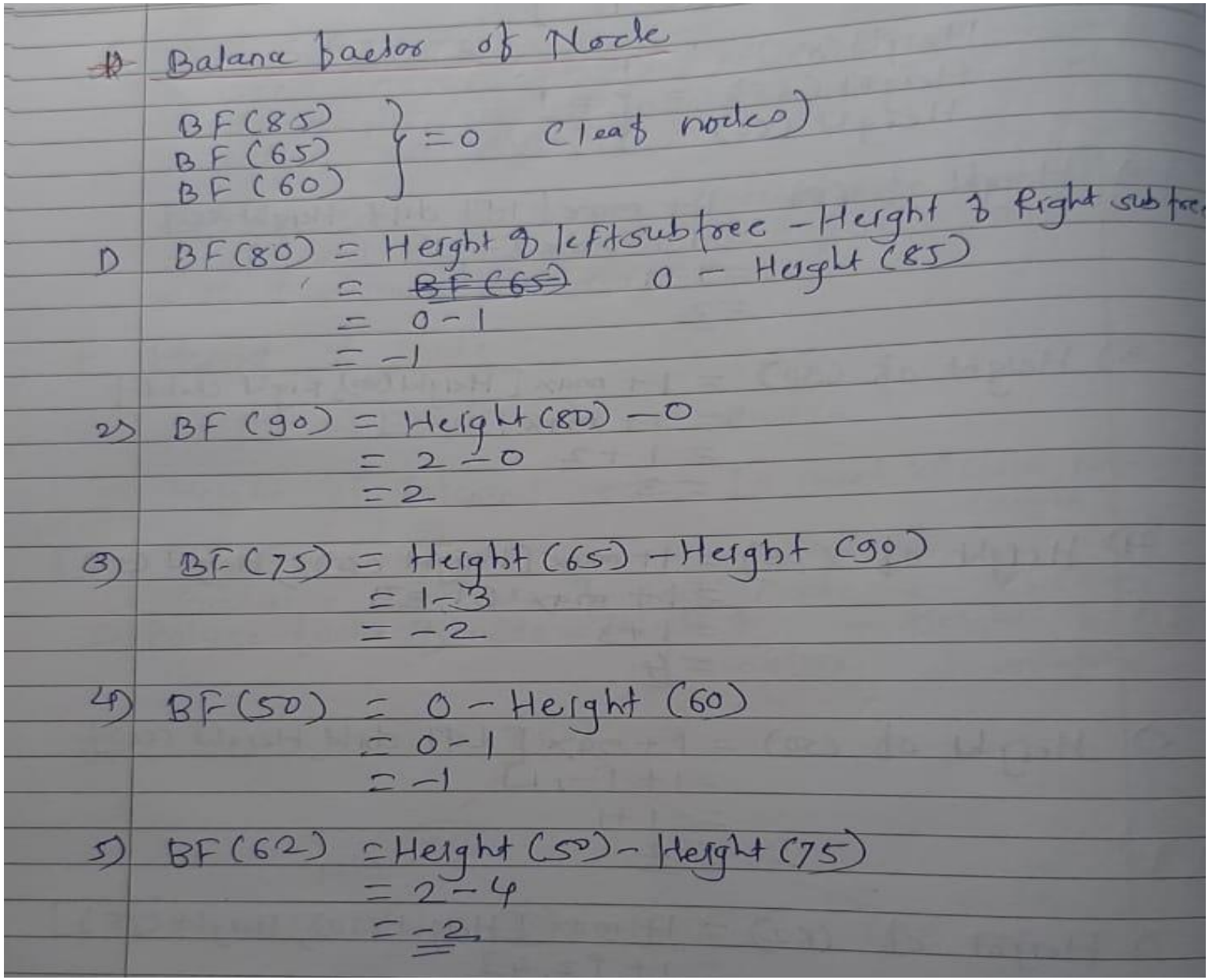
3) Height of (90) = 1 + max [Height(80), Right child]
= 1 + max [2, -]
= 1 + 2
= 3

4) Height of (75) = 1 + max [Height(65), Height(90)]
= 1 + max [1, 3]
= 1 + 3
= 4

5) Height of (50) = 1 + max [Left child, Height(60)]
= 1 + [-, 1]
= 1 + 1
= 2

7) Height of (62) = 1 + max [Height(50), Height(75)]
= 1 + [2, 4]
= 1 + 4
= 5

Balance Factor



AVL Rotations

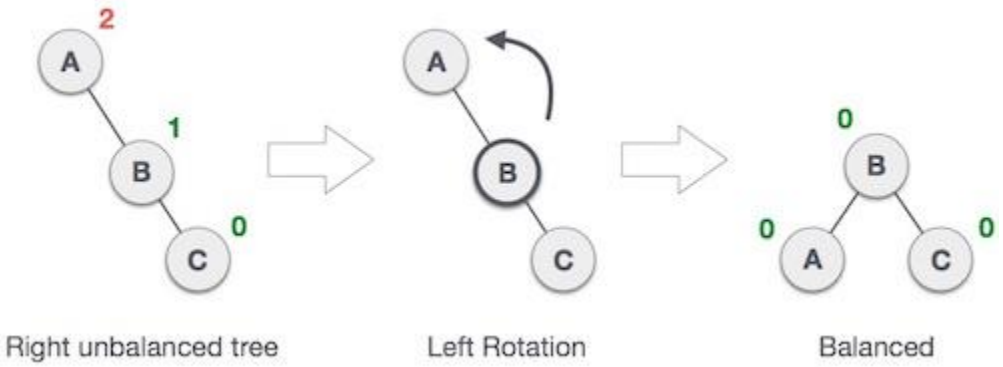
To balance itself, an AVL tree may perform the following four kinds of rotations −

1. Left rotation
2. Right rotation
3. Left-Right rotation
4. Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations.

1. Left Rotation

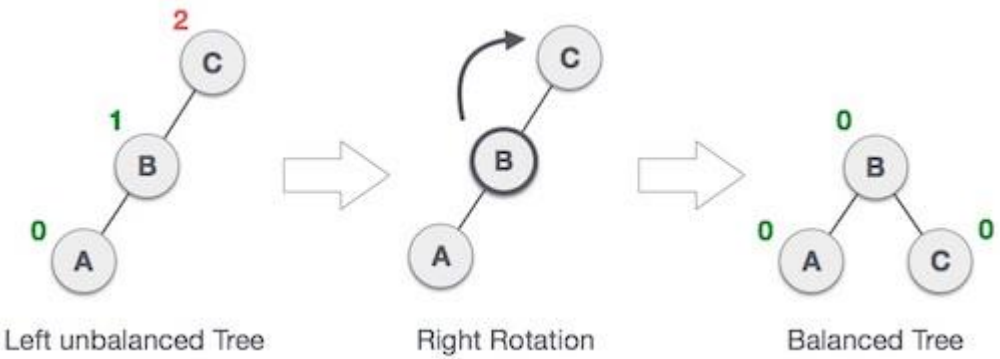
- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



- In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

2.Right Rotation

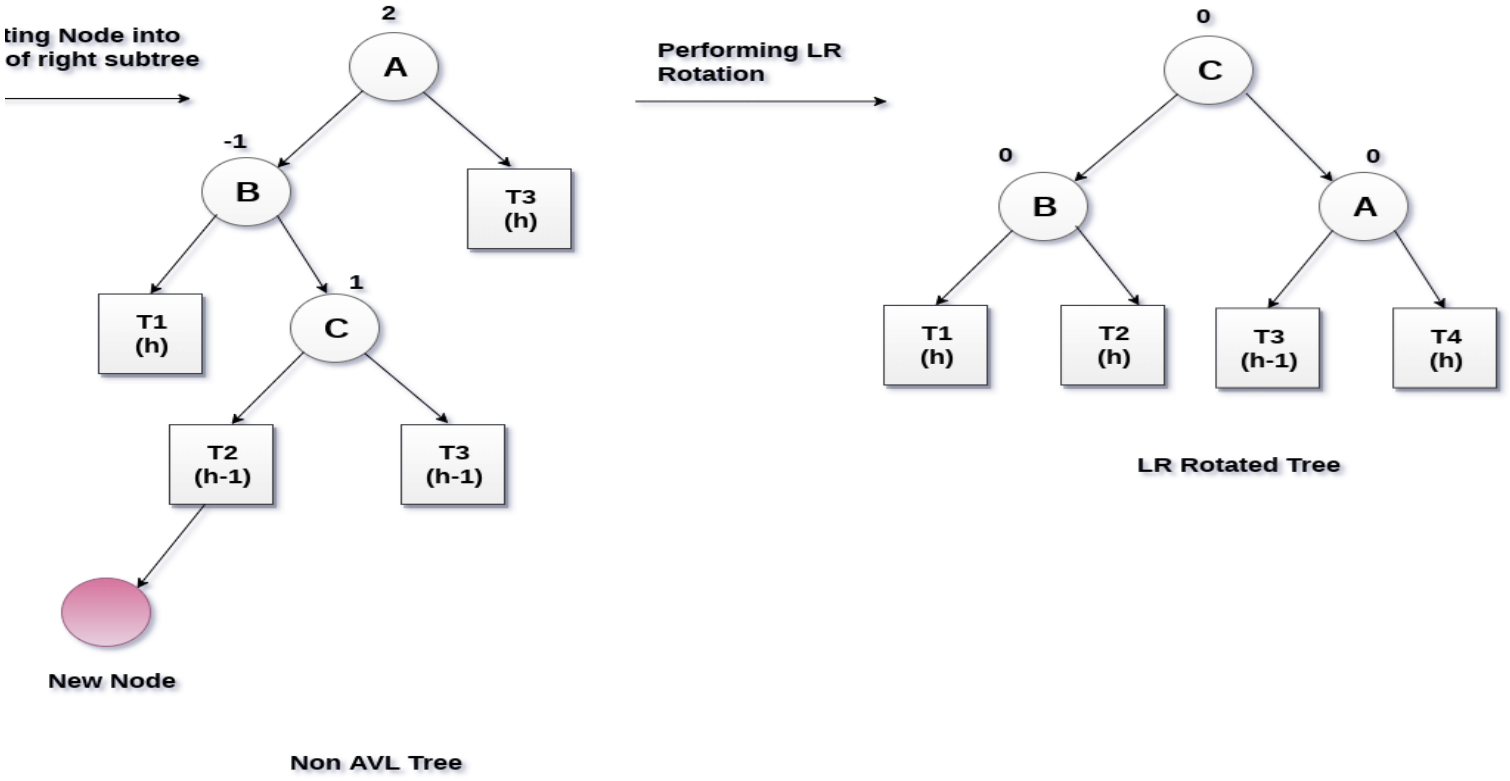
- AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

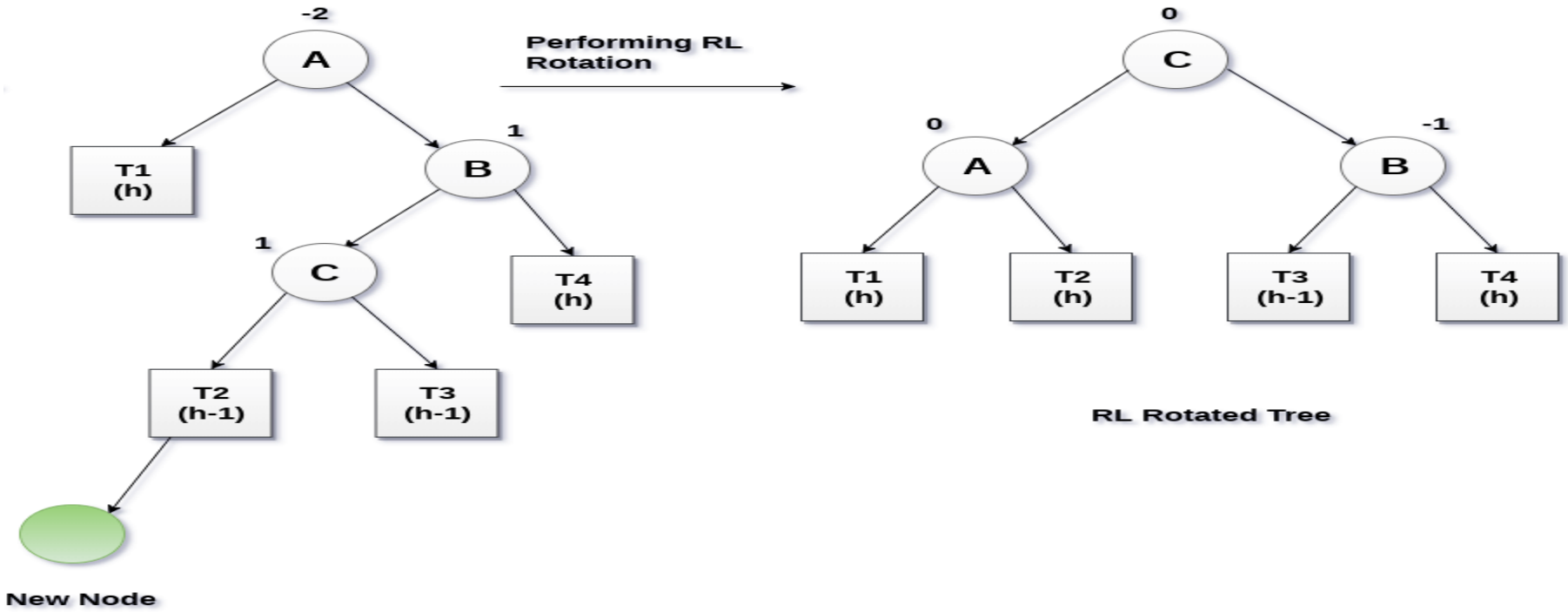
3. Left Right Rotation (LR Rotation)

- The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...

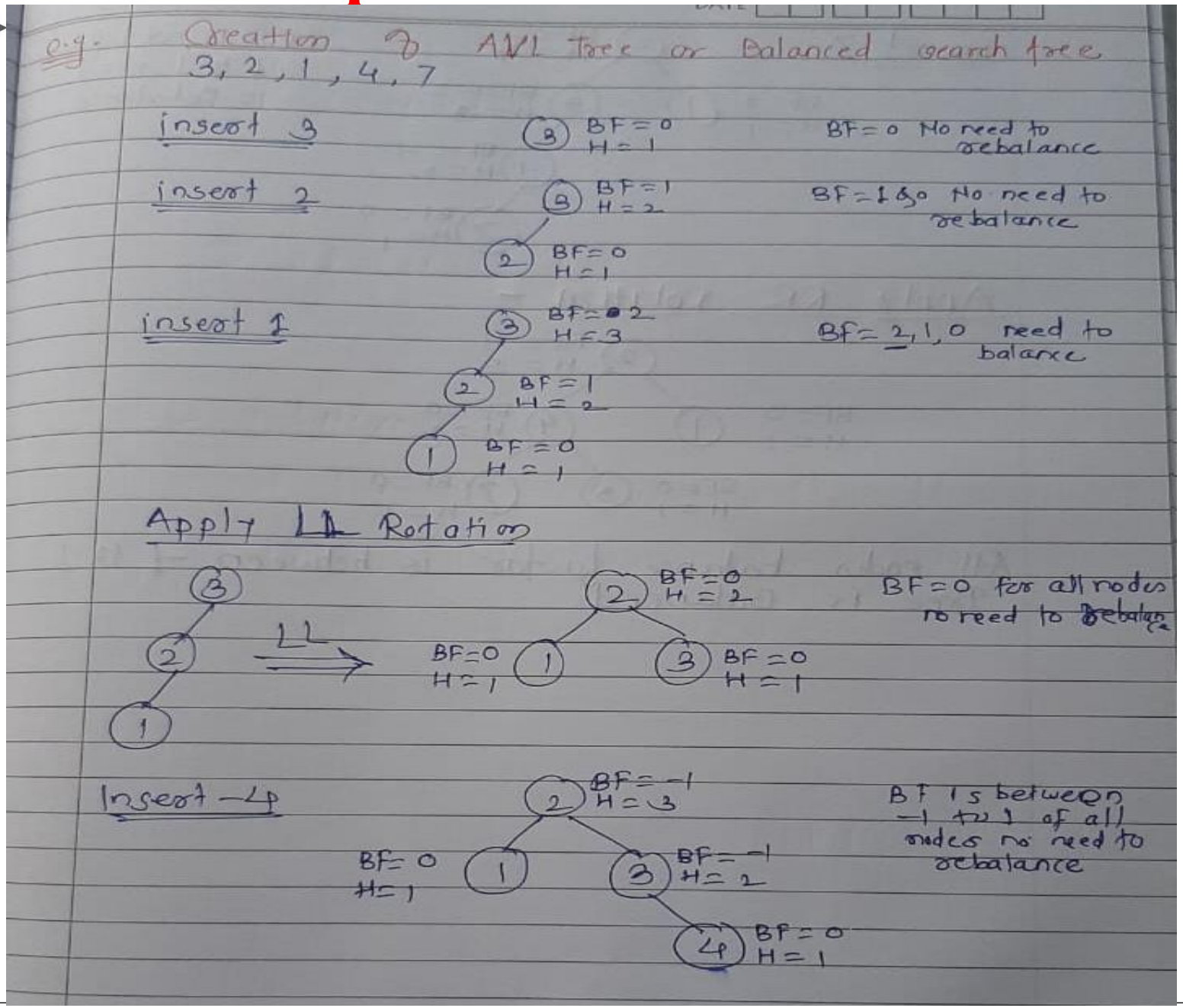


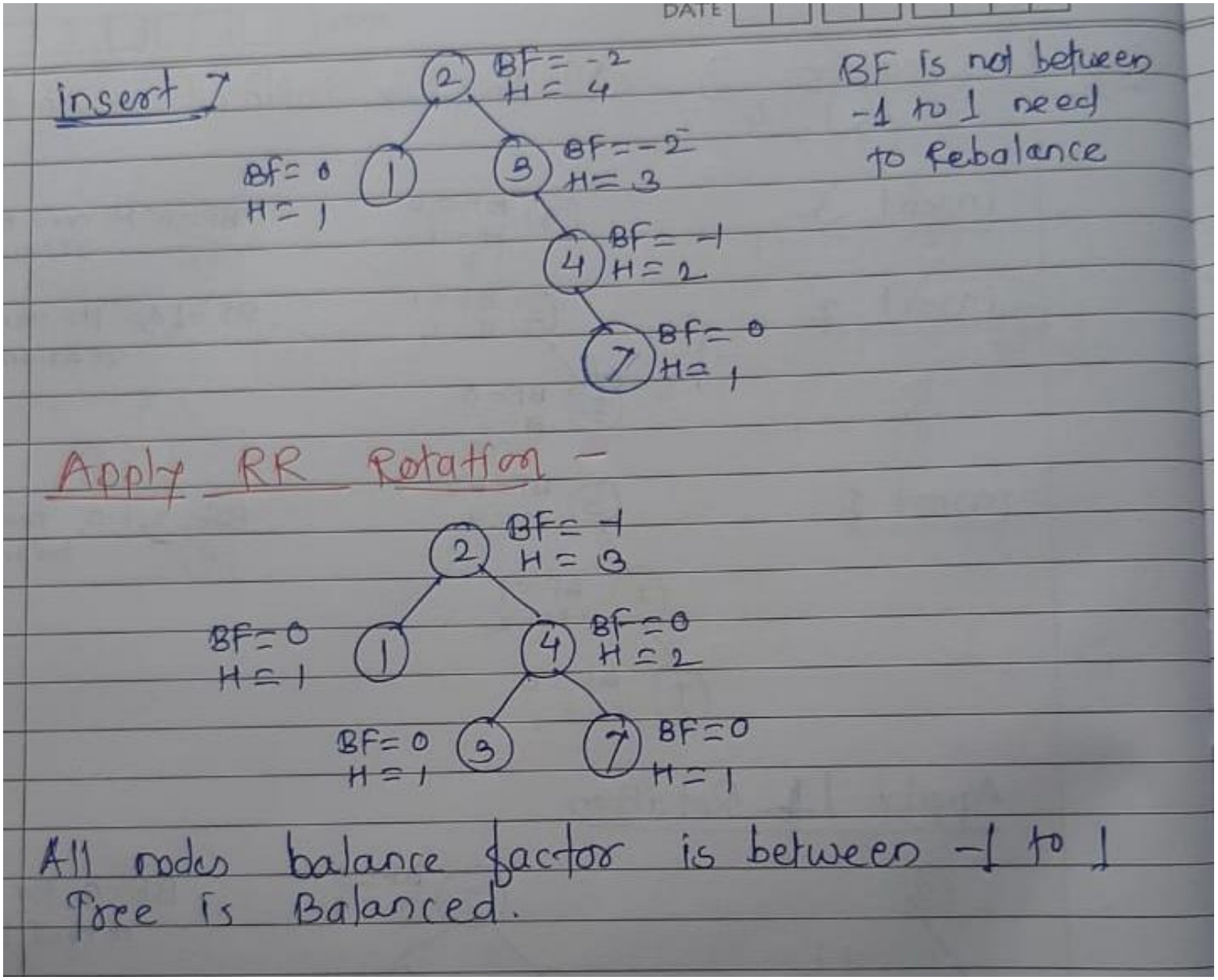
4. Right Left Rotation (RL Rotation)

- The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Example : To create AVL Tree




insert 7
2
BF = -2
H = 4
BF is not between -1 to 1 need to Rebalance
BF = 0
H = 1
1
3
BF = -2
H = 3
4
BF = -1
H = 2
7
BF = 0
H = 1
Apply RR Rotation -
2
BF = -1
H = 3
BF = 0
H = 1
1
4
BF = 0
H = 2
BF = 0
H = 1
3
7
BF = 0
H = 1
All nodes balance factor is between -1 to 1
Tree is Balanced.

7. Graphs

7.1 Definition of Graph

7.2 Basic Concepts of Graph

7.3 Representation of Graph

7.3.1 Adjacency Matrix

7.3.2 Adjacency List

7.4 Single Source shortest path algorithm-Dijkstra's algorithm.

7.5 Spanning Tree

7.6 Minimum Spanning Tree

7.6.1 Kruskal's Algorithm

7.6.2 PRIM's Algorithm

7.7 Graph Traversal

7.7.1 Breadth First Search (BFS)

7.7.2 Depth First Search (DFS)

What is Graph?

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices.

A graph is defined as follows...

Graph is a collection of vertices and arcs in which vertices are connected with arcs

Graph is a collection of nodes and edges in which nodes are connected with edges

Graph consists of two following components:

1. Vertices
2. Edges

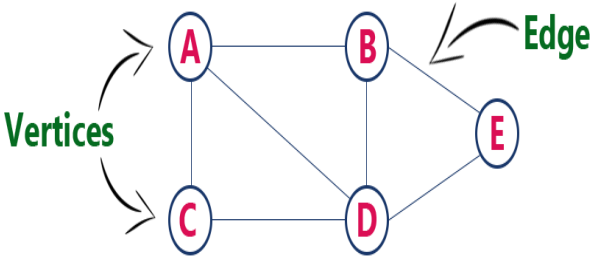
- Graph is a set of vertices (V) and set of edges (E).
- V is a finite number of vertices also called as nodes.
- E is a set of ordered pair of vertices representing edges.
- Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E}

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



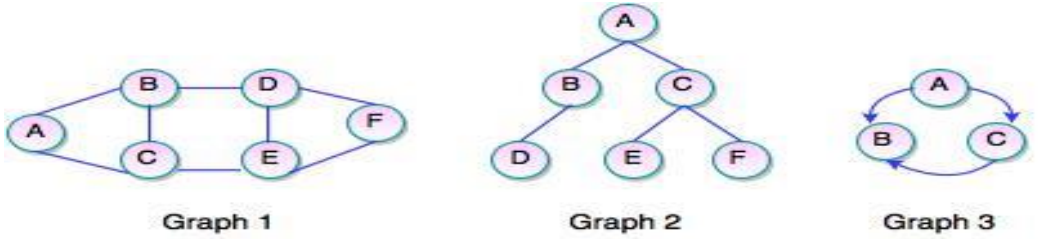


Fig. Graphs

Types of Graph

1. Directed Graph
2. Undirected Graph
3. Connected Graph
4. Disconnected Graph
5. Mixed Type Graph
6. Cyclic Graph
7. Acyclic Graph
8. Weighted Graph

1. Directed Graph :

A graph with only directed edges is called directed graph.

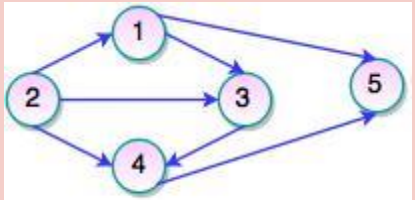


Fig. Directed Graph

2. Undirected Graph :

A graph without direction is called undirected graph.

In which each edge is not assigned a direction.

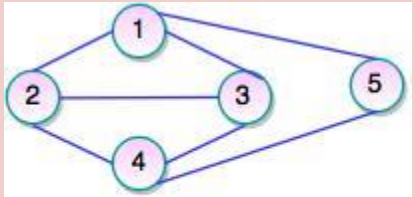
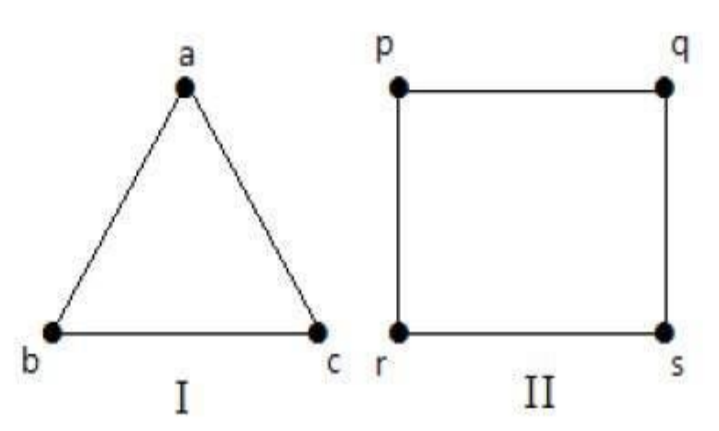


Fig. Undirected Graph

3. Connected Graph

A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

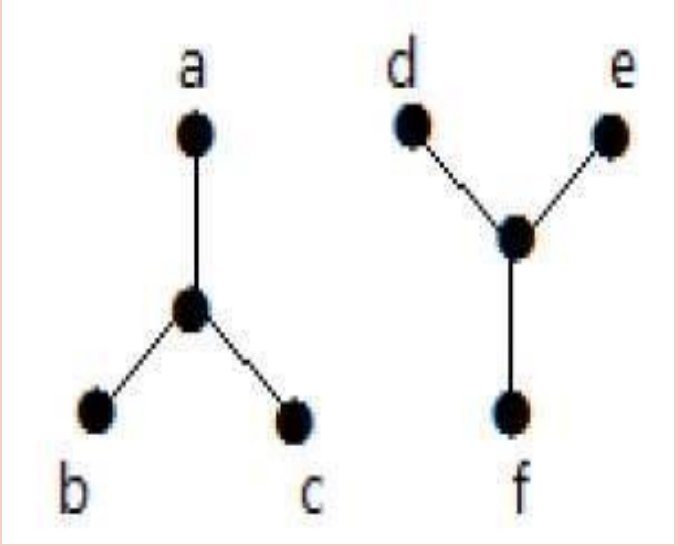


4. Disconnected Graph :

A graph G is disconnected,
if it does not contain at least two connected vertices.

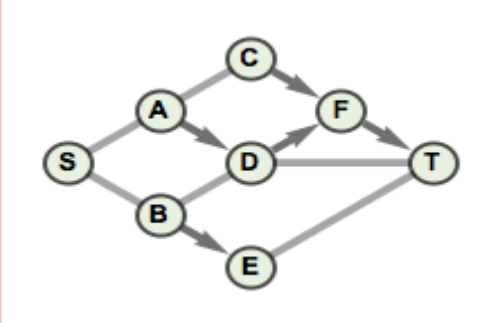
The following graph is an example of a Disconnected Graph, where there are two components, one with 'a', 'b', 'c' vertices and another with 'e', 'f', 'g'vertices.

The two components are independent and not connected to each other. Hence it is called disconnected graph.



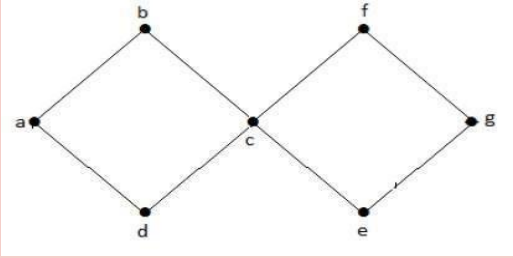
5. Mixed Type Graph

A graph in which some edges are directed and some edges are undirected such a graph is called as mixed type graph.



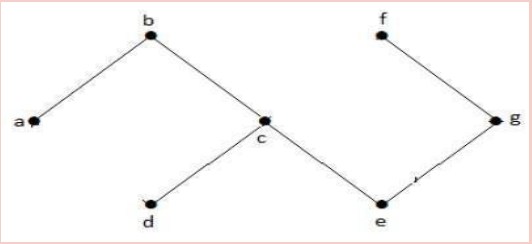
6. Cyclic Graph :

A graph **with at least one** cycle is called a cyclic graph.



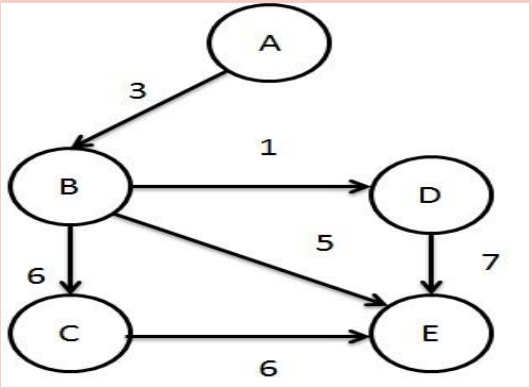
7. Acyclic Graph :

A graph **with no** cycle is called Acyclic graph.



8. Weighted Graph :

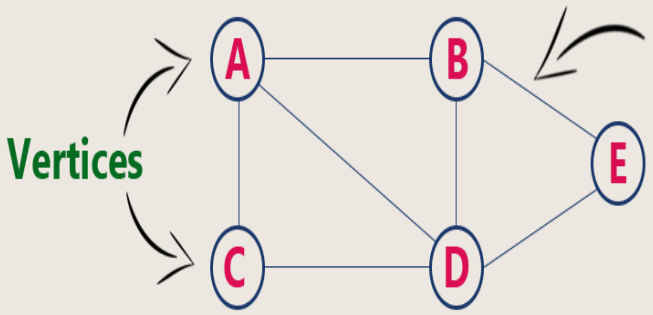
In a weighted graph, each edge has an associated numerical value, called the weight of the edge
Edge weights may represent distance, time, cost, etc.
A weighted graph can be directed or undirected.



Graph - Terminology

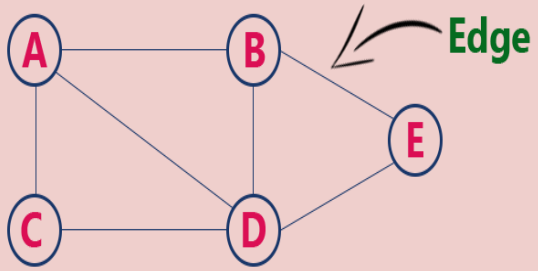
1. Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**.
In above example graph, A, B, C, D & E are known as vertices.



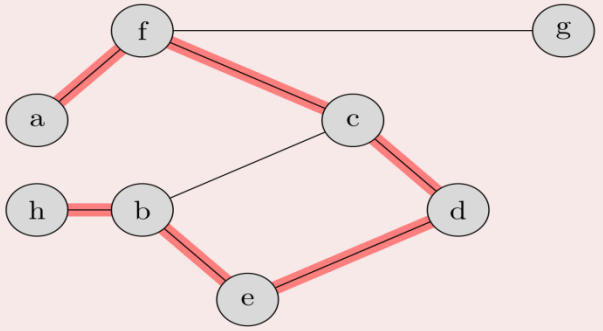
2. Edge

An edge is a connecting link between two vertices.
Edge is also known as **Arc**.



3. Path

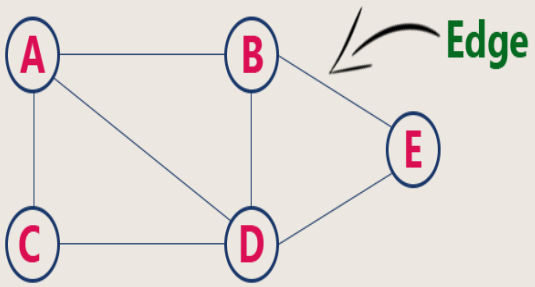
A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.



Graph - Terminology

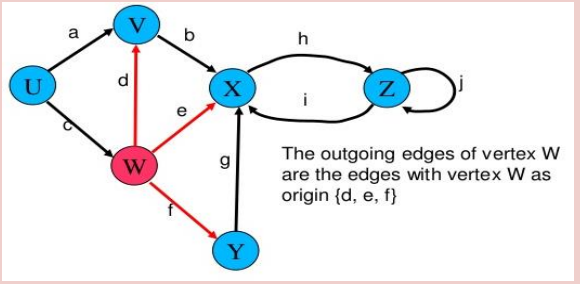
4. Adjacent node

Vertices B and E are said to be adjacent node .
If there is an edge between vertices B and E



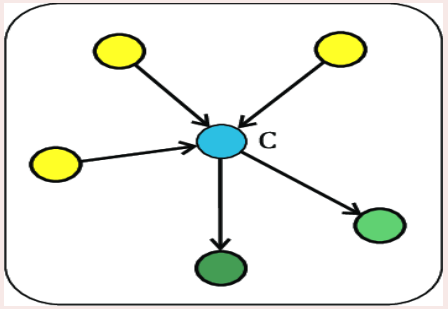
5. Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.



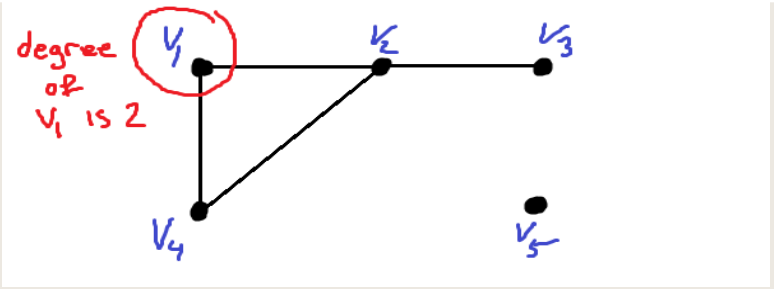
6. Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.



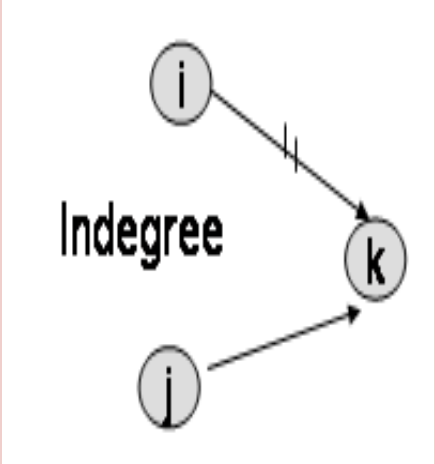
7. Degree

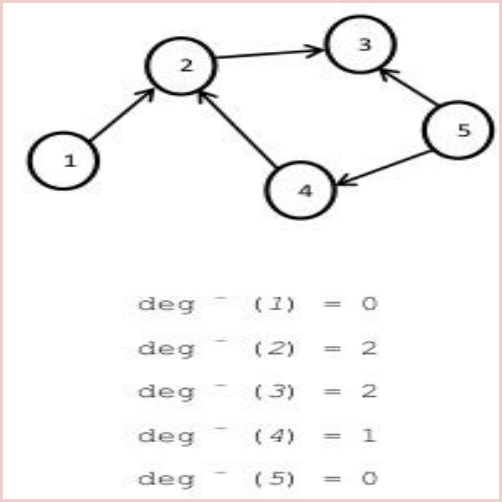
Total number of edges connected to a vertex is said to be degree of that vertex.



8. In degree

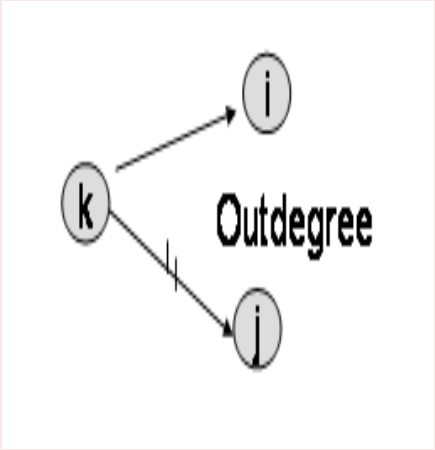
Total number of incoming edges connected to a vertex is said to be in degree of that vertex.

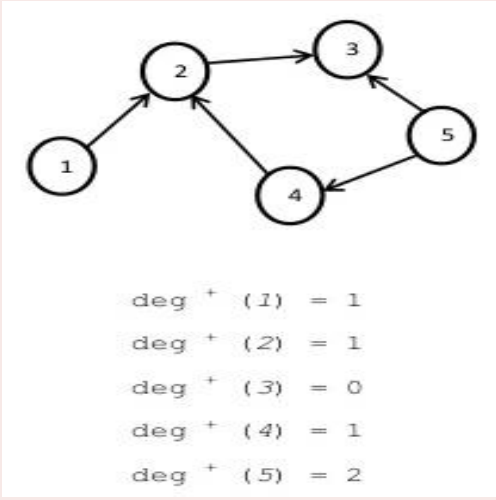




9. Out degree

Total number of outgoing edges connected to a vertex is said to be out degree of that vertex.

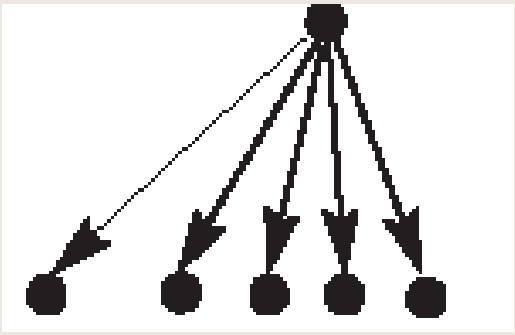




Graph - Terminology

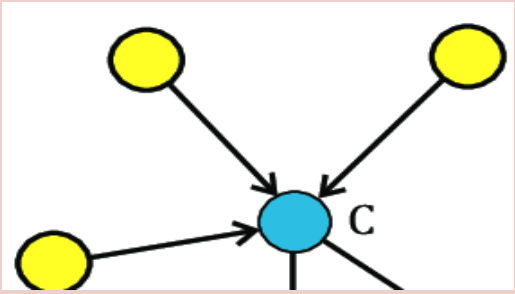
10. Source

A node which has only out going edges and no incoming edges is called source



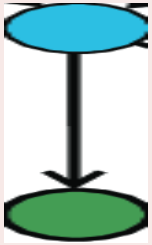
11. Sink

A node which has only incoming edges and no outgoing edges is called sink



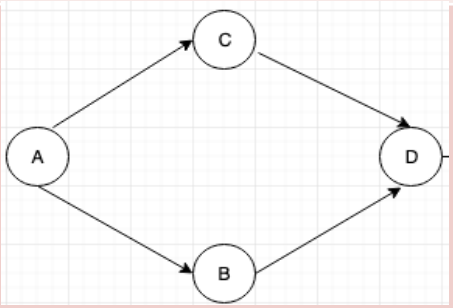
12. Pendant Node

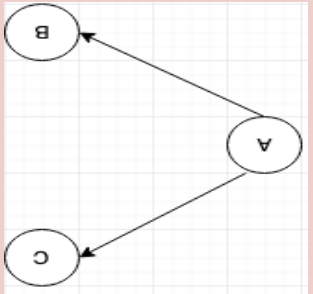
When indegree of node is one and out degree is zero then such a node is called pendant node vertex.



13. Articulation Point

If on removing the node the graph gets disconnected then that node is called articulation point..

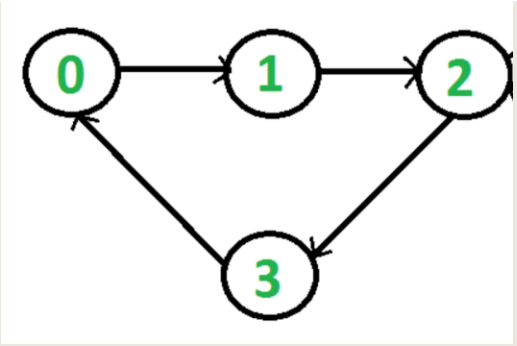




Graph - Terminology

14. Cycle

A path from node to itself is called cycle.
Thus cycle is a path in which the initial and final vertex is same

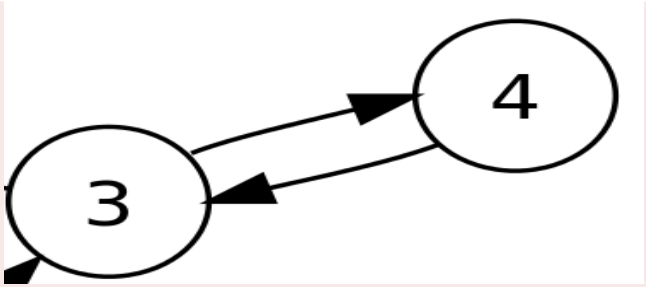


15. Sling or loop

An edge of a graph which joins a node to itself is called a sling or loop

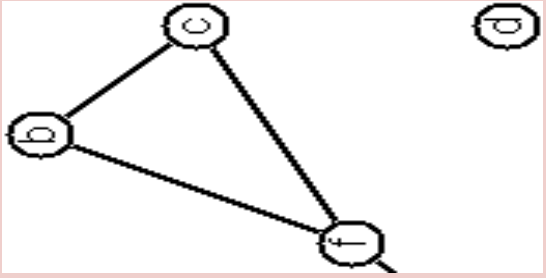
16. Parallel Edges

The two distinct edges between a pair to nodes which are opposite in direction are called as parallel edges.



17. Isolated node

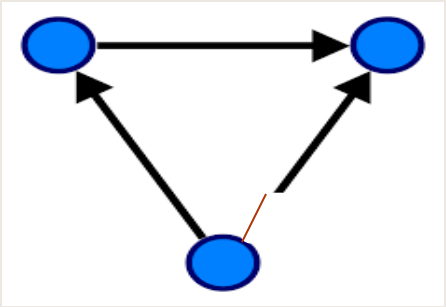
A node which is not an adjacent neighbor to any other node is called an isolated node.



Graph - Terminology

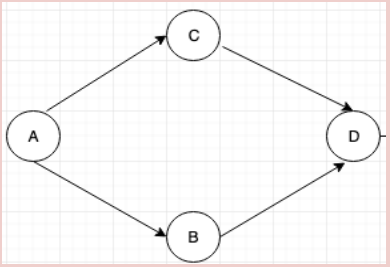
18. Directed Acyclic Graph

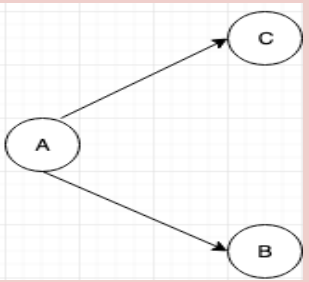
A directed graph with no cycle is called directed acyclic graph.



19. Sub graph

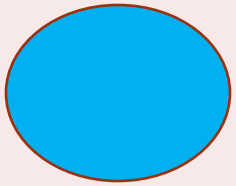
Sub graph is a graph G





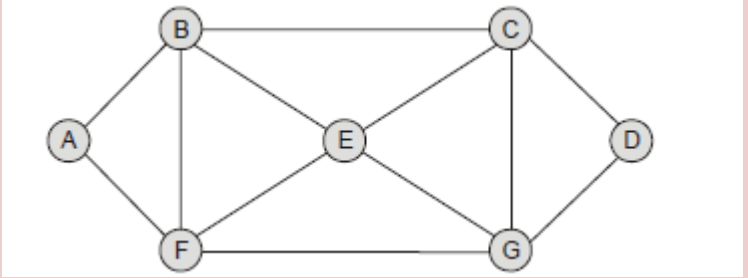
20. Isolated / Null Graph

A graph which has set of empty edges or is containing only isolated nodes is called NULL graph or or isolated graph



21. Biconnected Graph

A Biconnected graph is the graph which does not contain any articulation point.



Graph Representations

Graph data structure is represented using following representations…

1. **Adjacency Matrix**
2. **Adjacency List**

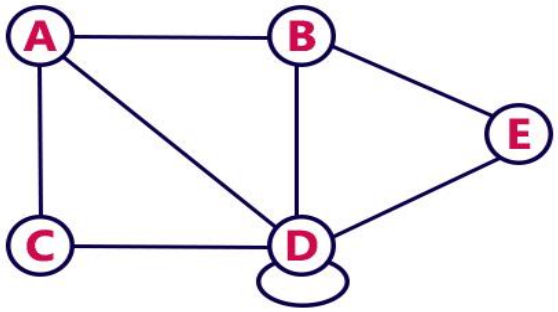
1. Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4.

In this matrix, both rows and columns represent vertices.

This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation…



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

Degree of Nodes

A=3

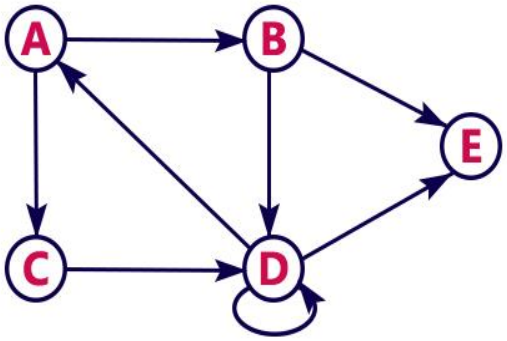
B=3

C=2

D=5

E=2

Directed graph representation...



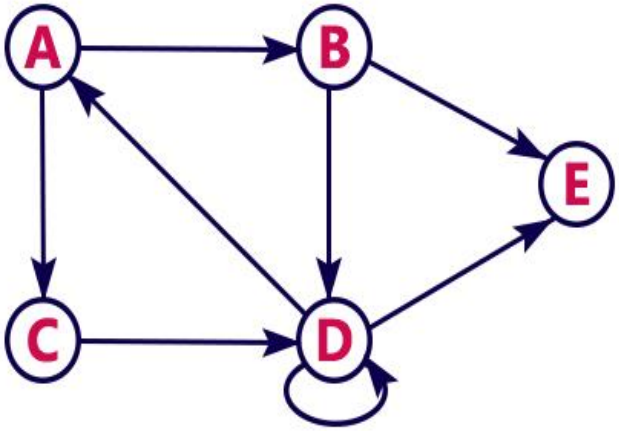
	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

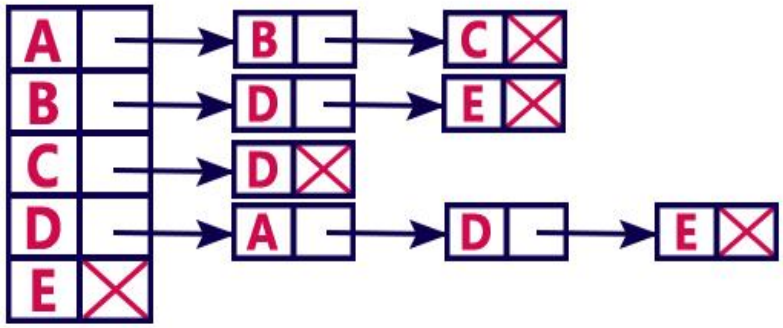
InDegree of Nodes	OutDegree of Nodes
A=1	A=2
B=1	B=2
C=1	C=1
D=3	D=3
E=2	E=0

2. Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

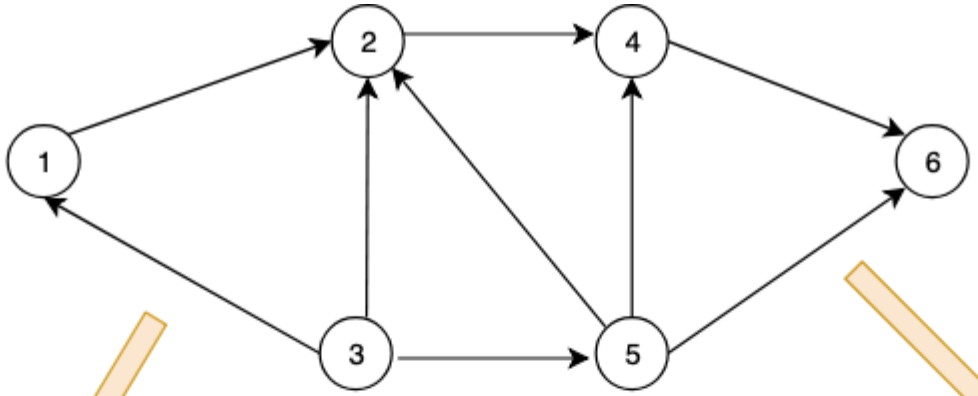
For example, consider the following directed graph representation implemented using linked list…





Example : Graph data structure is represented using following representations…

1. Adjacency Matrix 2. Adjacency List



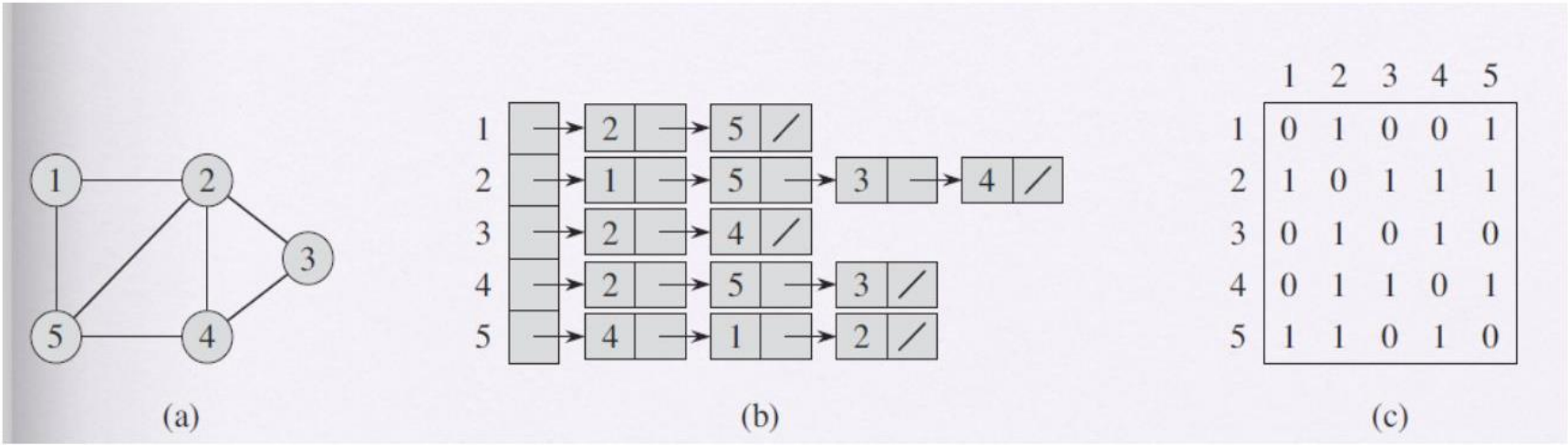
Adjacency List

Adjacency Matrix

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	0	0
3	1	1	0	0	1	0
4	0	0	0	0	0	1
5	0	1	0	1	0	1
6	0	0	0	0	0	0

Example : Graph data structure is represented using following representations…

1. **Adjacency Matrix 2. Adjacency List**



Example : Graph data structure is represented using following representations…

1. **Adjacency Matrix 2. Adjacency List**

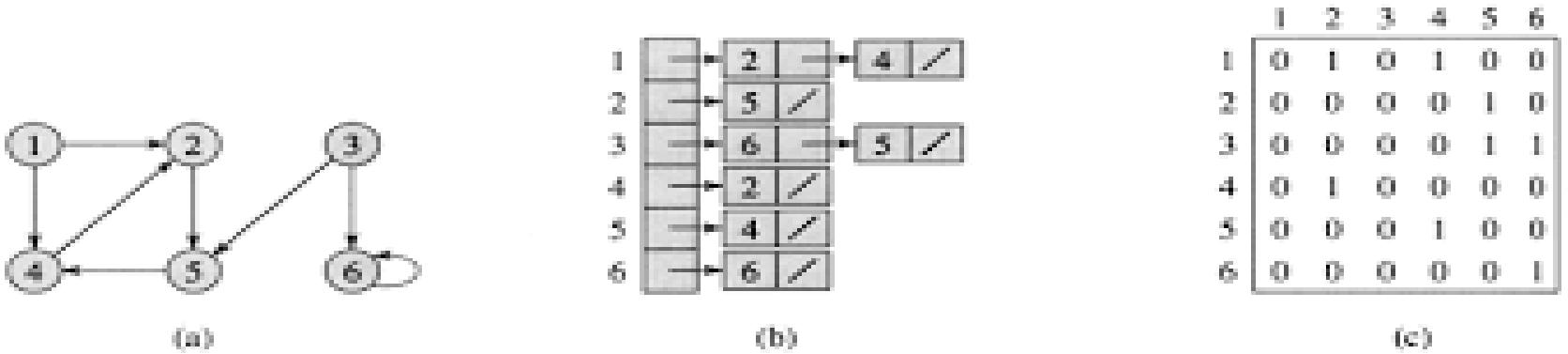
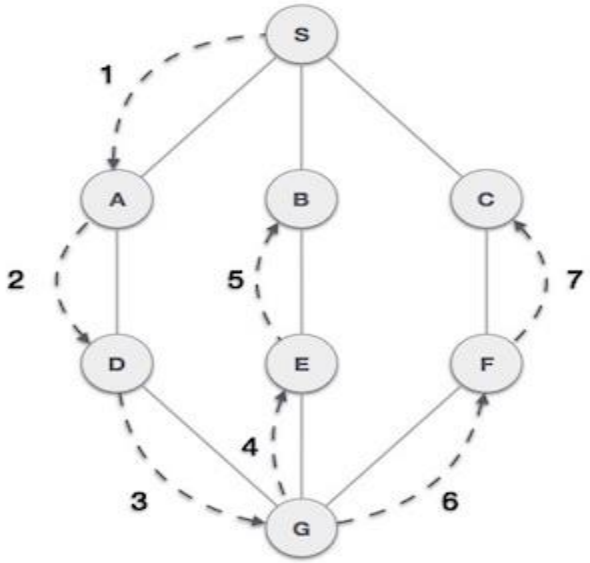


Figure 23.2 Two representations of a directed graph. (a) A directed graph G having six vertices and eight edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

1. Depth First Search (DFS) algorithm (using Stack)

Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

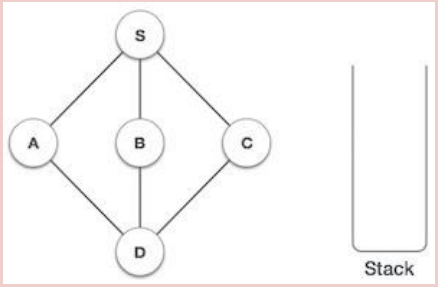
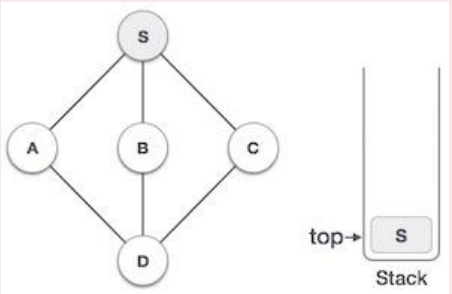
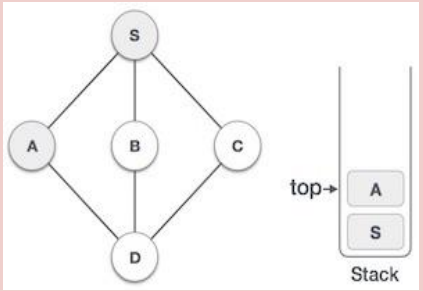
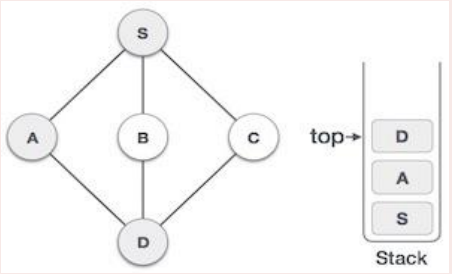


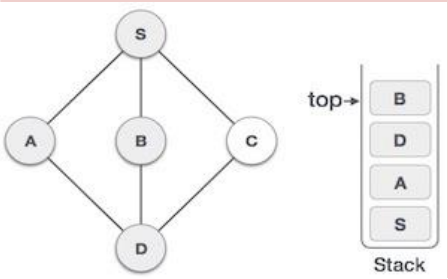
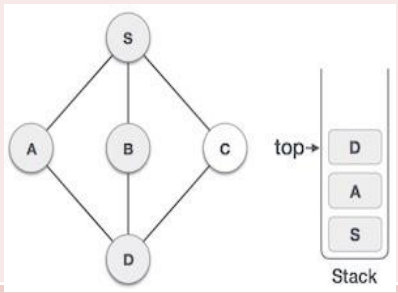
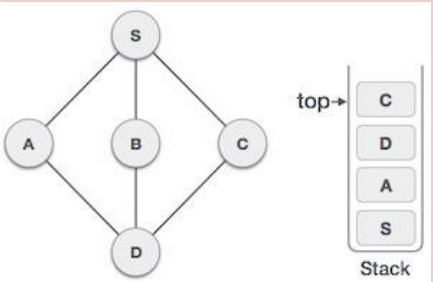
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

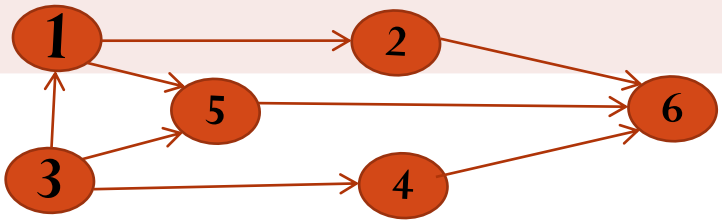
Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

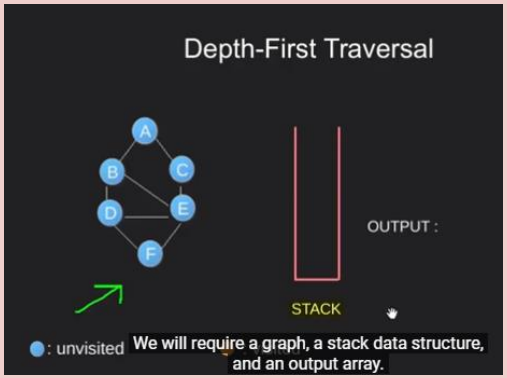
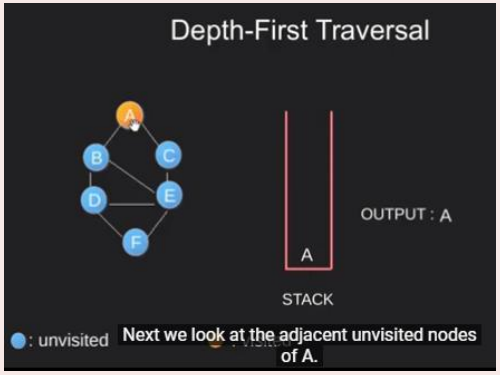
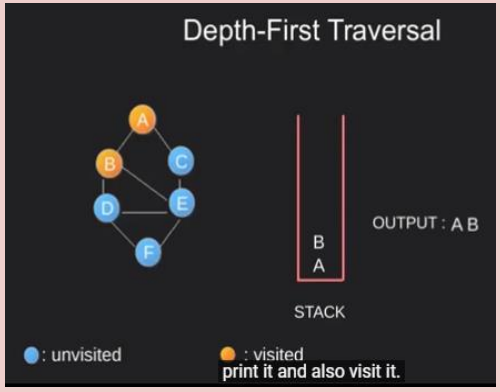
Step	Traversal	Description
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

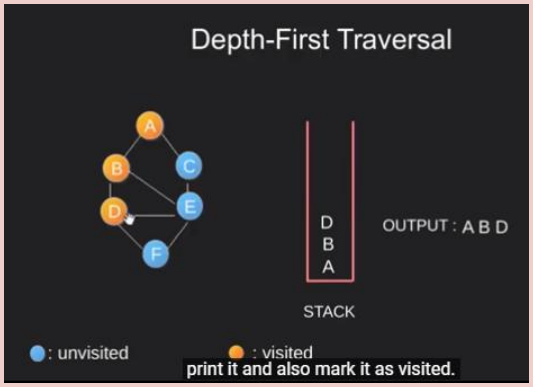
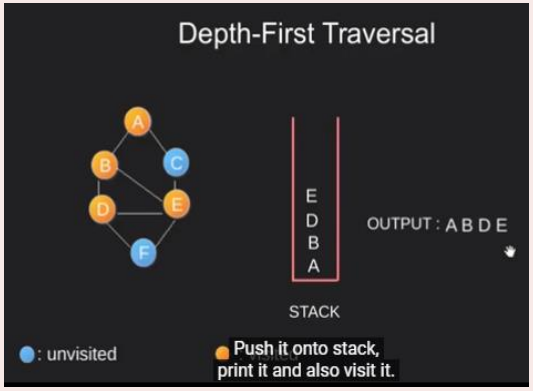
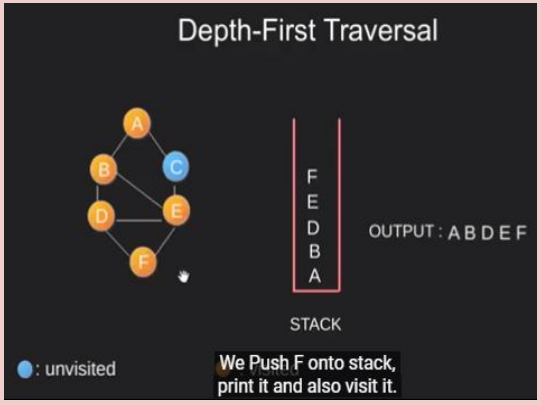
As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty **DFS Traversing = S,A,D,C**

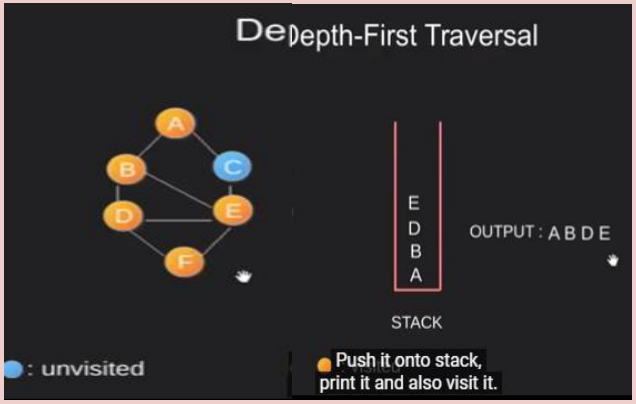
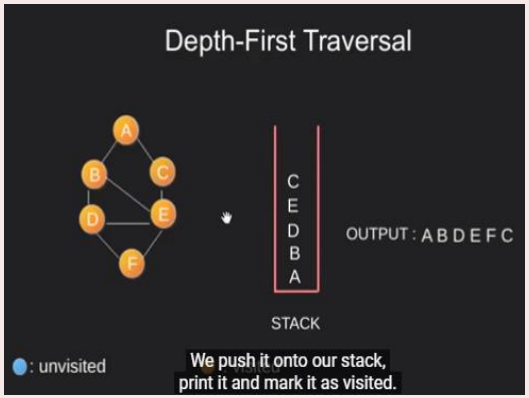


Solve Example DFS

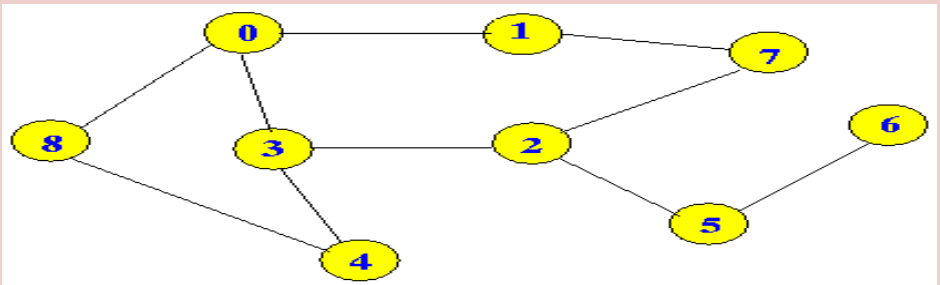
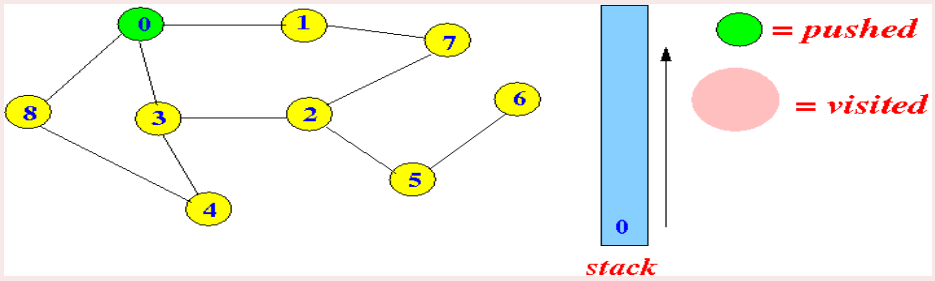
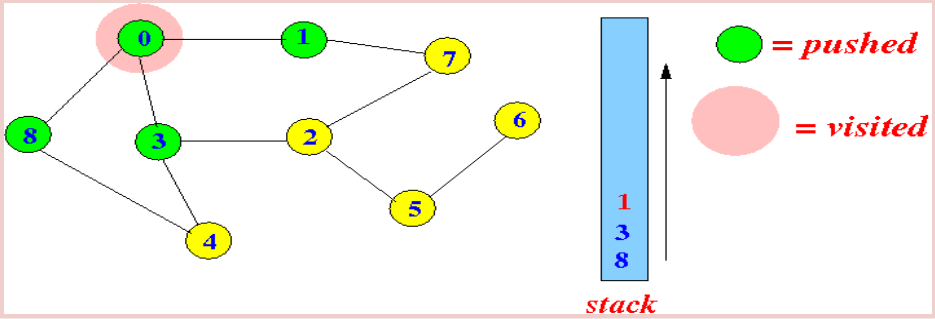
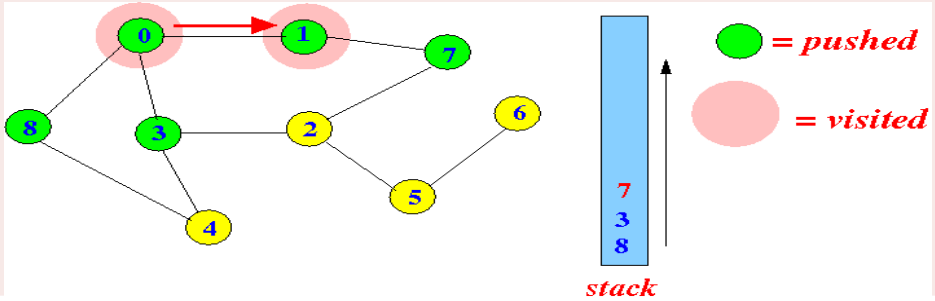
DFS Traversing =

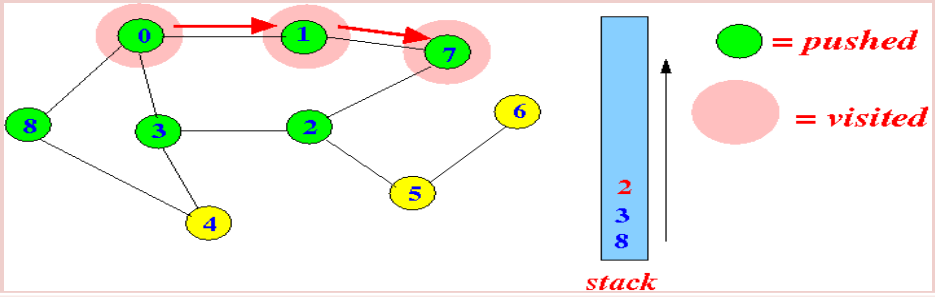
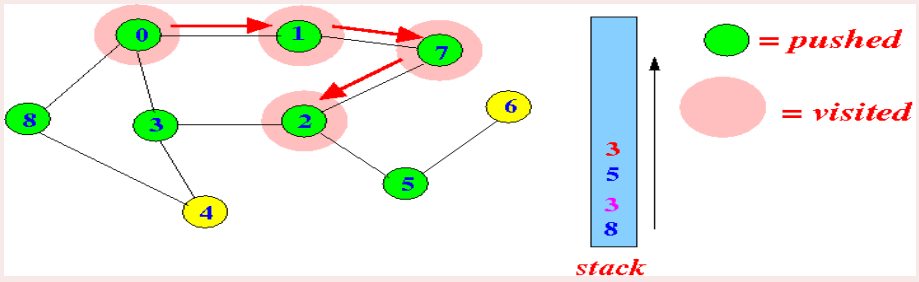
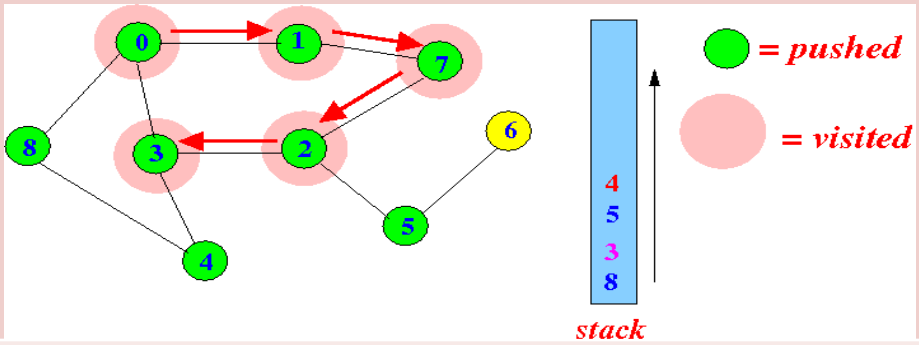
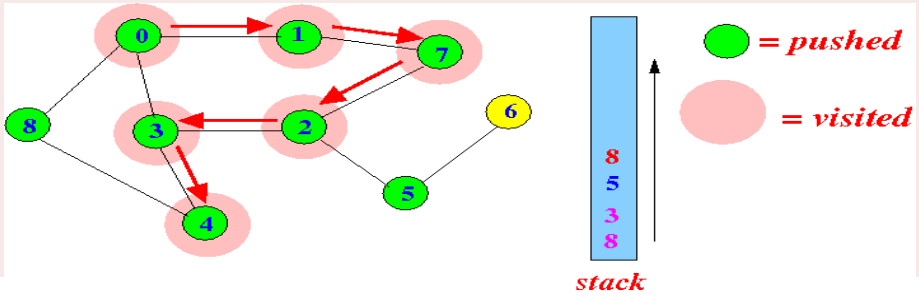
Step	Traversal	Description
1		Initialize the stack.
2		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. We have two nodes (B and C) and we can pick any of them. For this example, we shall take the node in an alphabetical order is B
3		We choose **B**, mark it as visited and put onto the stack. Explore any unvisited adjacent node from B. We have two nodes (D and E) and we can pick any of them. For this example, we shall take the node in an alphabetical order is D

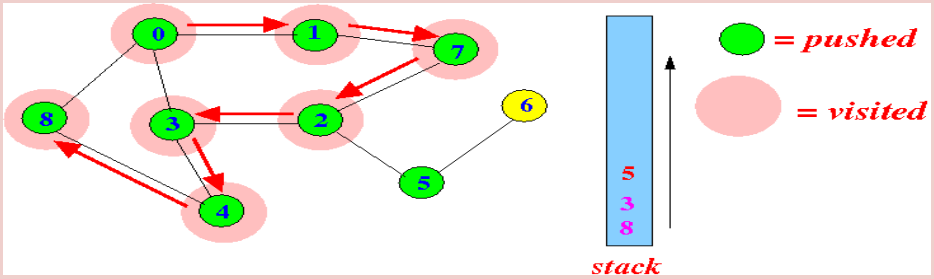
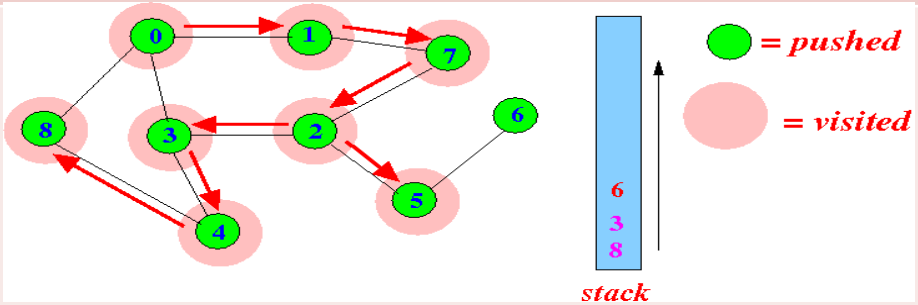
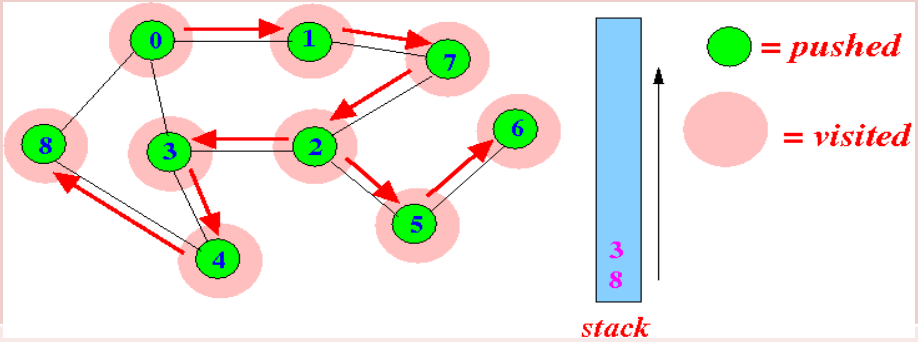
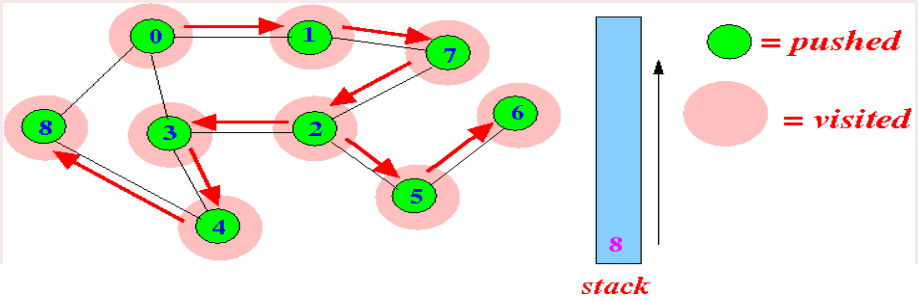
Step	Traversal	Description
5		We choose D, mark it as visited and put onto the stack. Explore any unvisited adjacent node from D . We have two nodes (E and F) and we can pick any of them. For this example, we shall take the node in an alphabetical order is E
6		We choose E, mark it as visited and put onto the stack. Explore any unvisited adjacent node from E . We have two nodes (F and C) and we can pick any of them. For this example, we shall take the node F
7		We choose F, mark it as visited and put onto the stack. Explore any unvisited adjacent node from F . Here **F** does not have any unvisited adjacent node. So, we pop **F** from the stack.

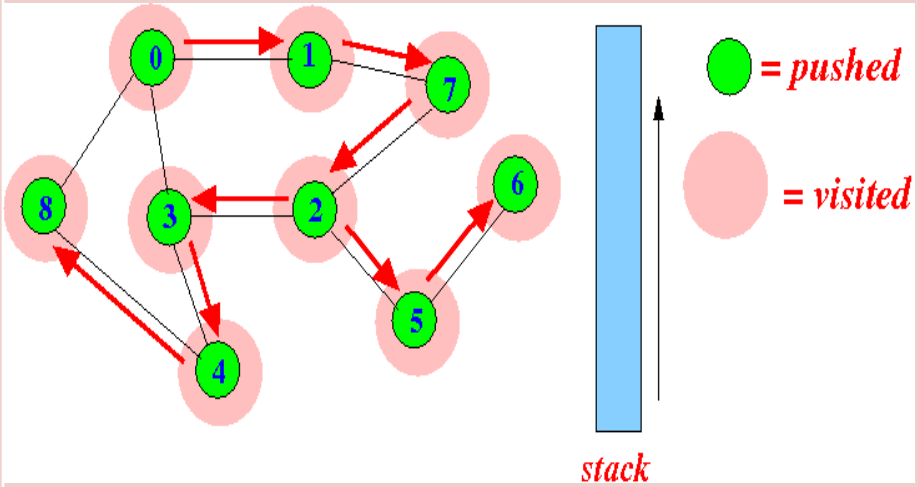
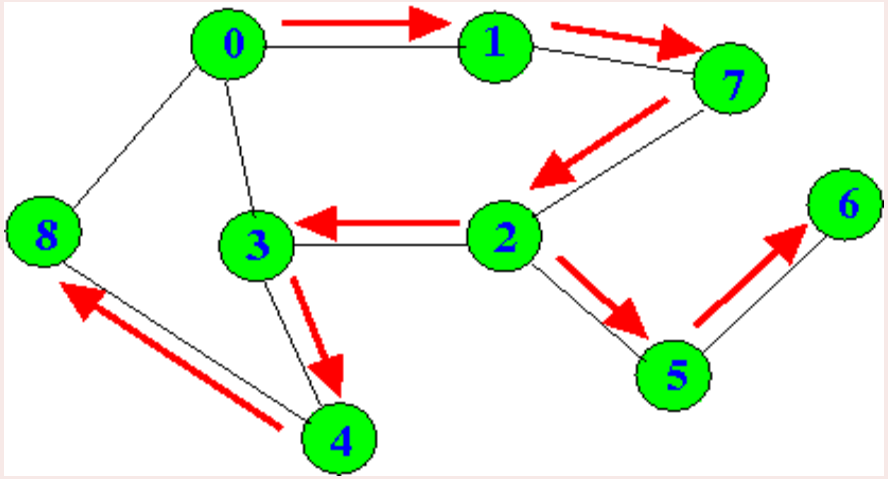
Step	Traversal	Description
5		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find C to be on the top of the stack.
6		Only unvisited adjacent node is from E is **C** now. So we visit **C**, mark it as visited and put it onto the stack.
7	DFS Traversal = A B C D E F	

Example of the DFS algorithm

Step	Traversal	Description
1		
2		**Initial state: node 0** is *pushed*
3		**State *after* visiting 0** **Push** the ***unvisited*** **neighbor nodes**: **8, 3, 1** (I used the reverse order to visit smaller node id first) **Next, visit** the ***top*** **node** in the **stack**: **1**
4		**State *after* visiting 1** **Push** the ***unvisited*** **neighbor nodes**: **7** **Next, visit** the ***top*** **node** in the **stack**: **7**

Step	Traversal	Description
5		**State *after* visiting 7** **Push** the ***unvisited*** **neighbor nodes**: 2 **Next, visit** the ***top*** **node** in the **stack**: 2
6		**State *after* visiting 2** **Push** the ***unvisited*** **neighbor nodes**: **5, 3** (Note: 3 is pushed again, and the previous value will be cancelled later -- as we will see) **Next, visit** the ***top*** **node** in the **stack**: **3**
7		**State *after* visiting 3** **Push** the ***unvisited*** **neighbor nodes**: 4 **Next, visit** the ***top*** **node** in the **stack**: 4
8		**State *after* visiting 4** **Push** the ***unvisited*** **neighbor nodes**: 8 (Note: 8 is pushed again, and the previous value will be cancelled later -- as we will see) **Next, visit** the ***top*** **node** in the **stack**: 8

Step	Traversal	Description
9		**State *after* visiting 8** **Push** the ***unvisited*** **neighbor nodes: none** **Next, visit** the ***top*** **node** in the **stack: 5**
10		**State *after* visiting 5** **Push** the ***unvisited*** **neighbor nodes: 6** **Next, visit** the ***top*** **node** in the **stack: 6**
11		**State *after* visiting 6** **Push** the ***unvisited*** **neighbor nodes: none** **Next, visit** the ***top*** **node** in the **stack: 3** **3 is visited : skip**
12		**Next, visit** the ***top*** **node** in the **stack: 8** **8 is visited : skip**

Step	Traversal	Description
13		**DONE** (The **stack** hasbecome **empty**)
14		**Traversal order**

Single Source shortest path algorithm- Dijkstra's algorithm.

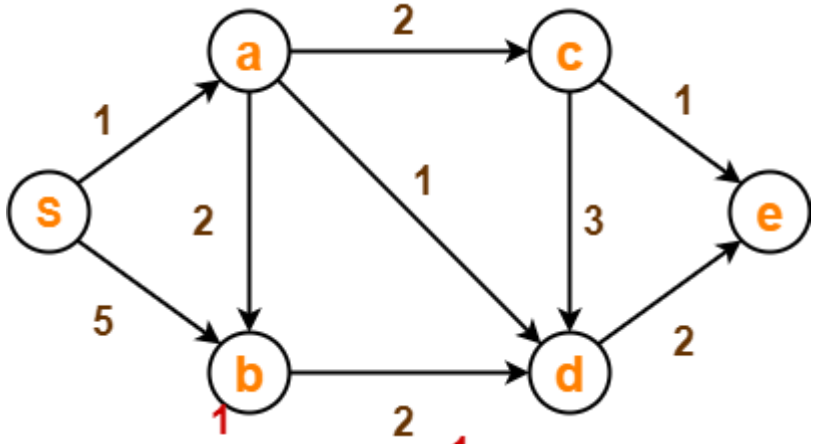
- Dijkstra Algorithm is a very famous greedy algorithm.
- It is used for solving the single source shortest path problem.
- It computes the shortest path from one particular source node to all other remaining nodes of the graph.

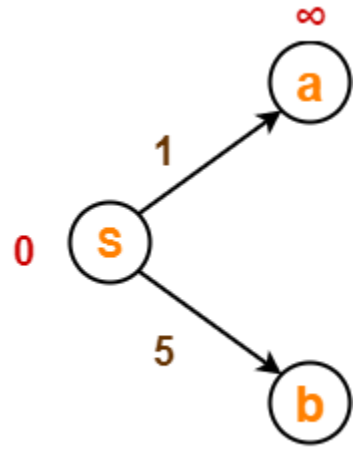
Conditions :

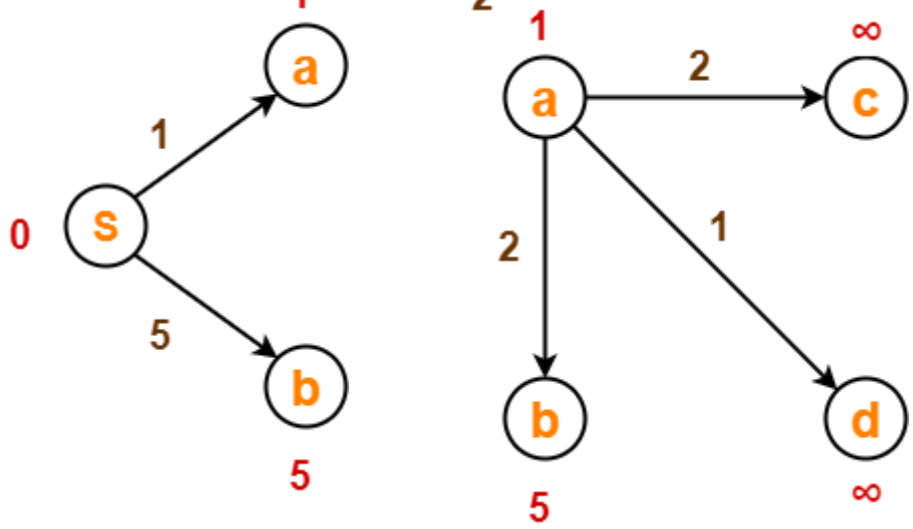
- It is important to note the following points regarding Dijkstra Algorithm-
- Dijkstra algorithm works only for connected graphs.
- Dijkstra algorithm works only for those graphs that do not contain any negative weight edge.
- The actual Dijkstra algorithm does not output the shortest paths.
- It only provides the value or cost of the shortest paths.
- By making minor modifications in the actual algorithm, the shortest paths can be easily obtained.
- Dijkstra algorithm works for directed as well as undirected graphs.

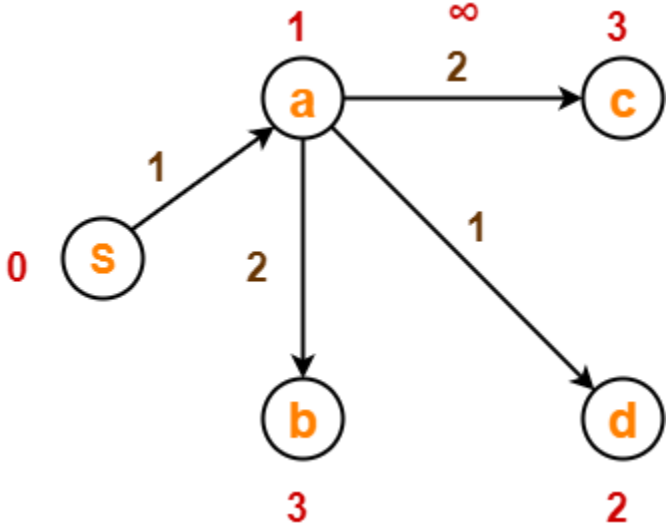
Problem-

Using Dijkstra's Algorithm, find the shortest distance from source vertex 'S' to remaining vertices in the following graph-



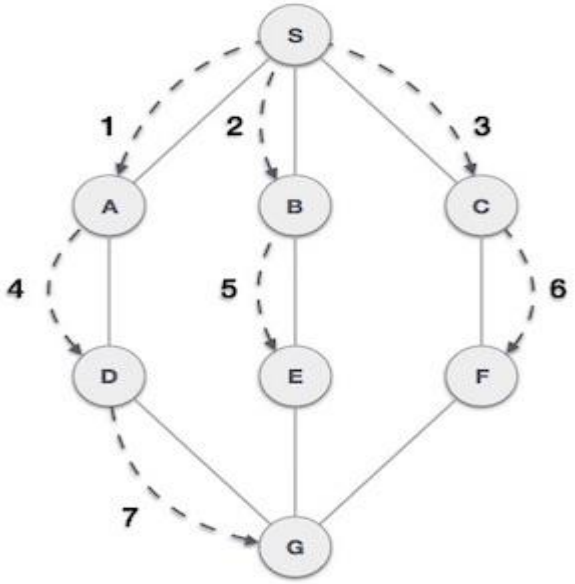






2. Breadth First Search (BFS) Algorithm(using QUEUE)

Breadth First Search (BFS) algorithm that begins at the root node and explorers all the neighbors node. That for each of those nearest nodes, it explorers their unexplored nodes and so on until it finds the goal.

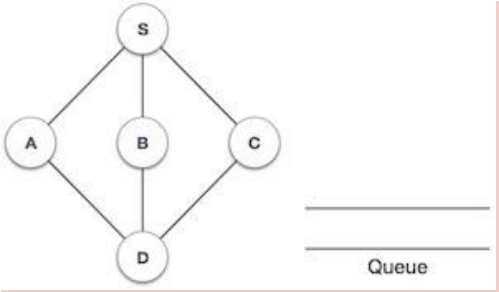
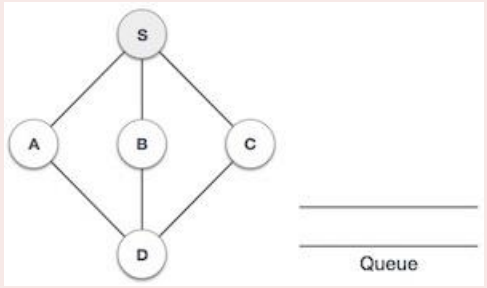
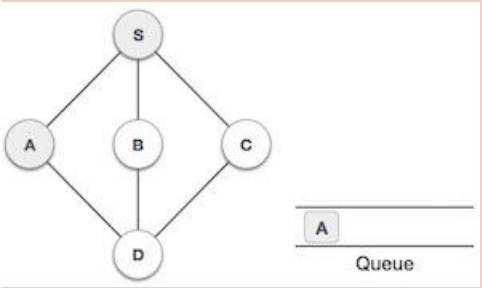
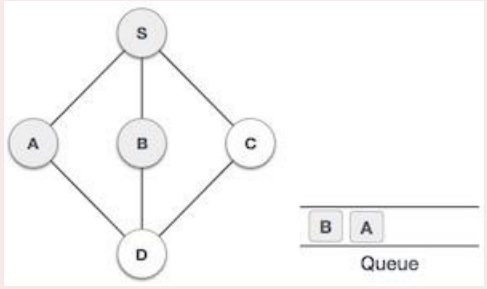


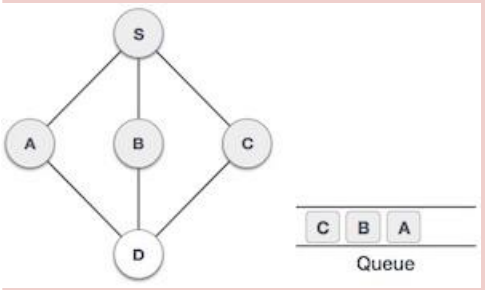
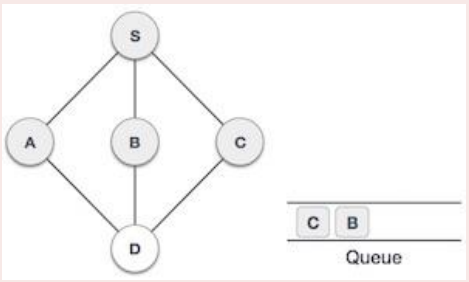
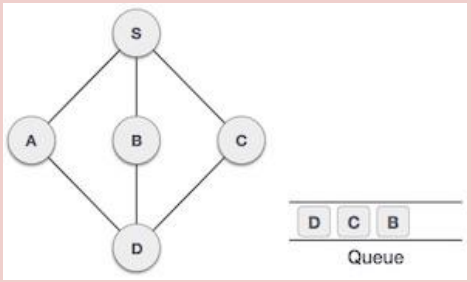
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

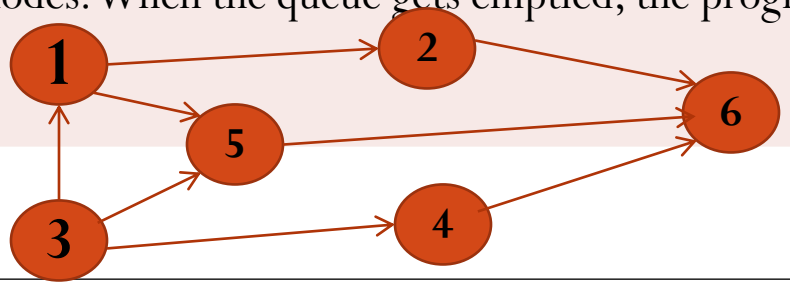
Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.

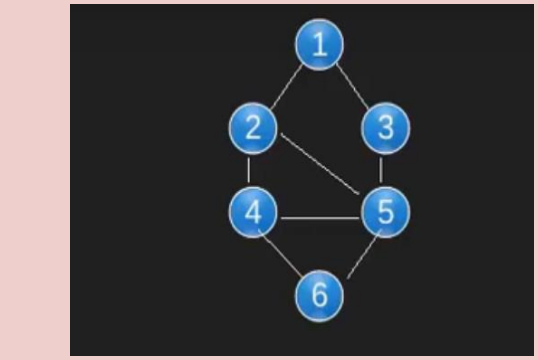
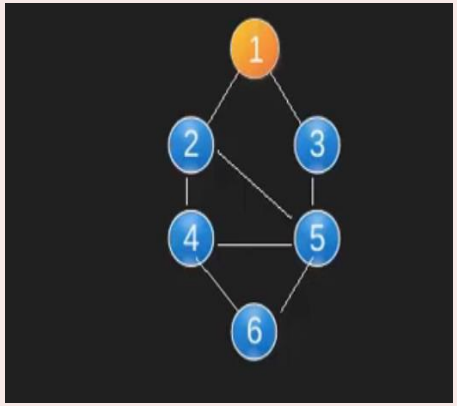
Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

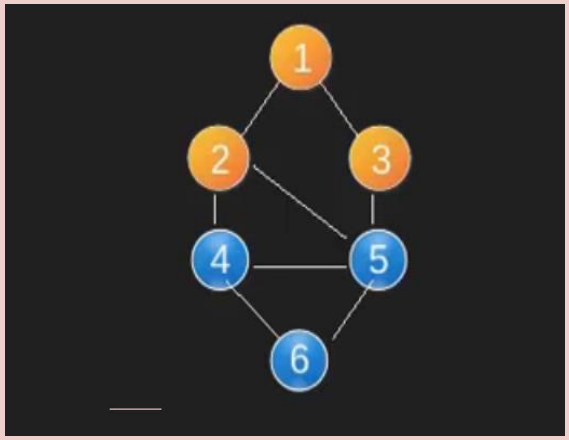
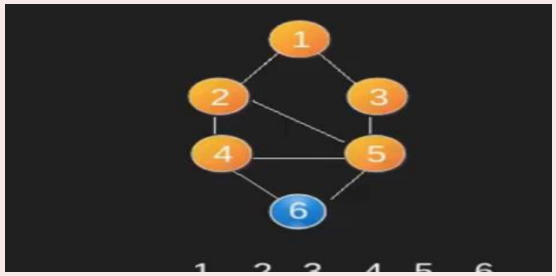
Step	Traversal	Description
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

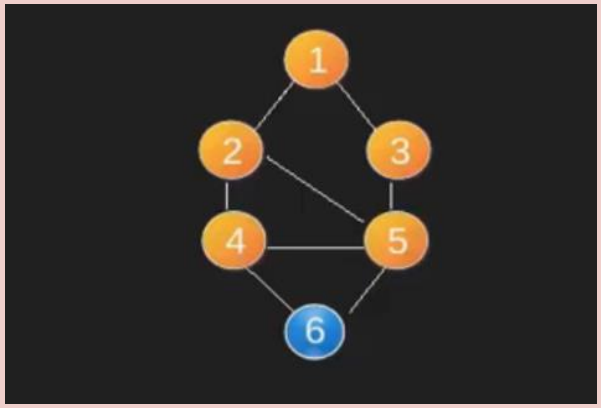
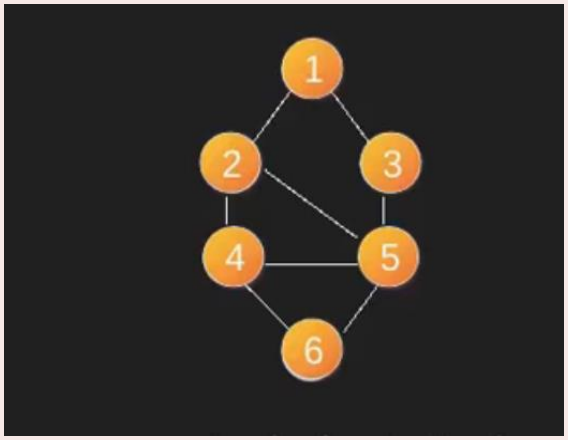
At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.



Solve Example BFS

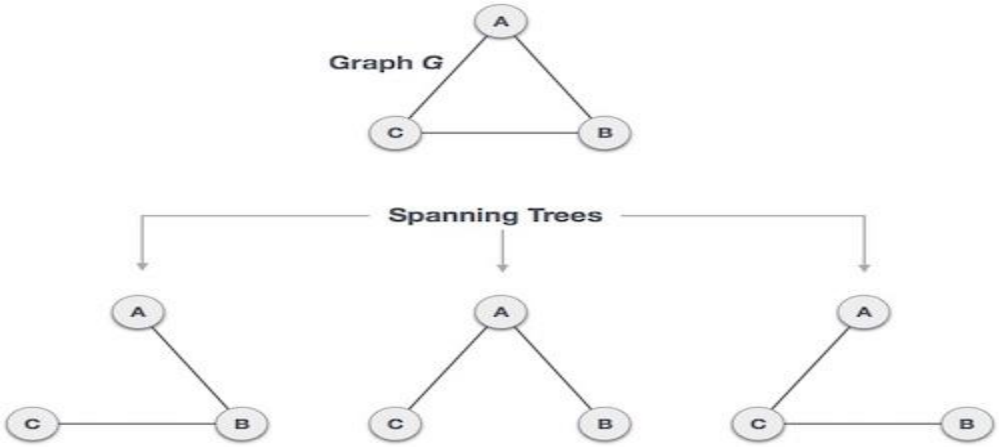
Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting 1 (starting node), and mark it as visited and put into Queue

Step	Traversal	Description
1	 2 | 3	We then see an unvisited adjacent node from 1 is (2,3) We Put both the node in queue and delete 1 from queue
2	 3 | 4 | 5	We then see an unvisited adjacent node from 2 is (4,5) We Put both the node in queue and delete 2 from queue

Step	Traversal	Description
1	 5 | 6	We then see an unvisited adjacent node from 4 is (6) We Put that in queue and delete 3 and 4 from queue
2		There is no unvisited node in queue so we remove 5 and 6. BFS Traversal is = 1 2 3 4 5 6

Spanning Tree

- A spanning tree of graph G=(V,E) is connected sub graph of g having all vertices of G and no cycle in it.
- Spanning Tree doesn't contain cycle.
- It is not unique, there can be more than one spanning tree of a graph.
- If graph is connected then it will always have a spanning tree.



Applications of Spanning Tree :

1. Routing of packets in network :

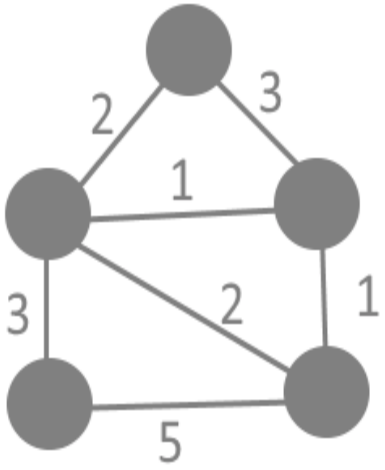
A node can represent the router located in a city and the link between router can be represented using an edge . A spanning tree can be represents a network with minimum no. of links

2. Electrical Network :

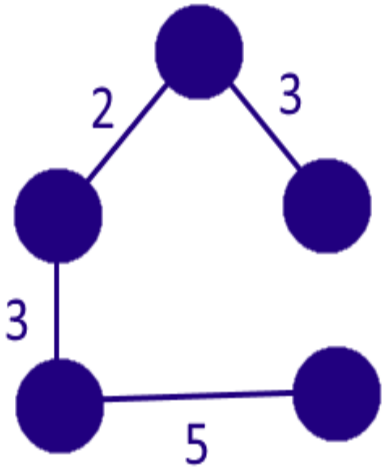
A spanning tree can be used to obtain an independent set of circuit equation for an electric network.

Minimal Spanning Tree

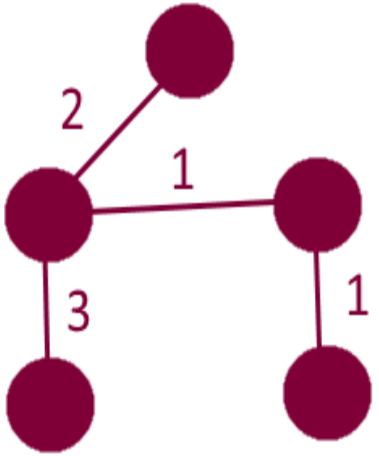
- A spanning tree T of graph G where the sum of weights of all edges in T is minimum is called the minimal cost spanning tree or minimal spanning tree of G



Graph



Spanning Tree
Cost = 13



Minimum Spanning Tree, Cost = 7

Minimal Spanning Tree

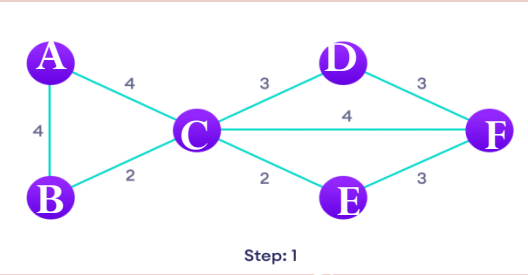
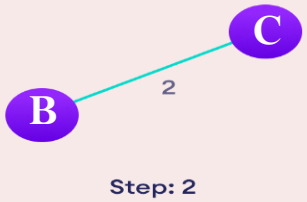
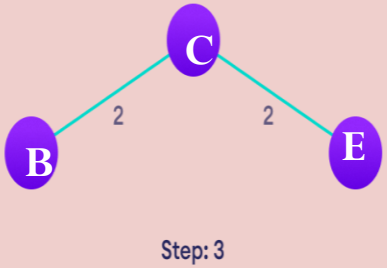
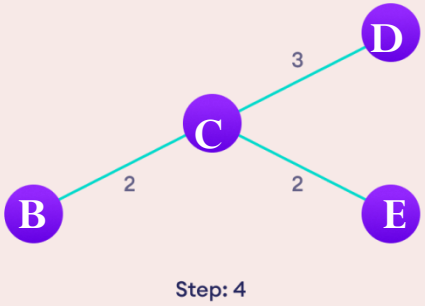
- There are two popular algorithms to calculate minimal cost spanning tree of a weighted undirected graph

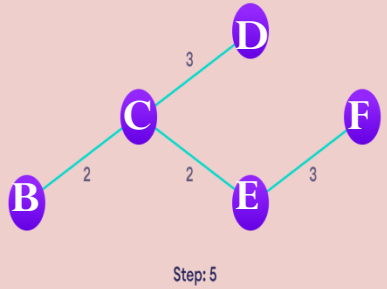
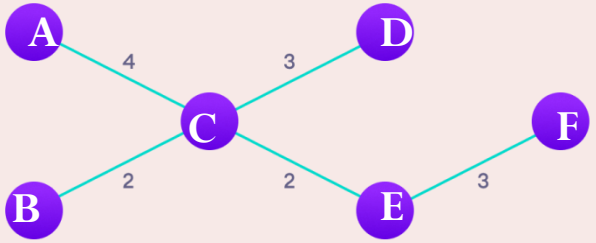
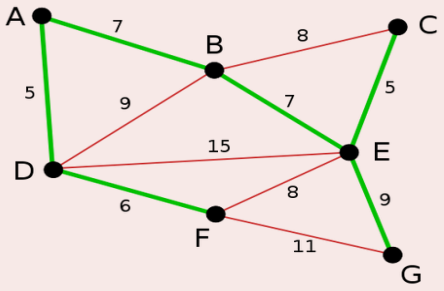
1 Kruskal's Algorithm

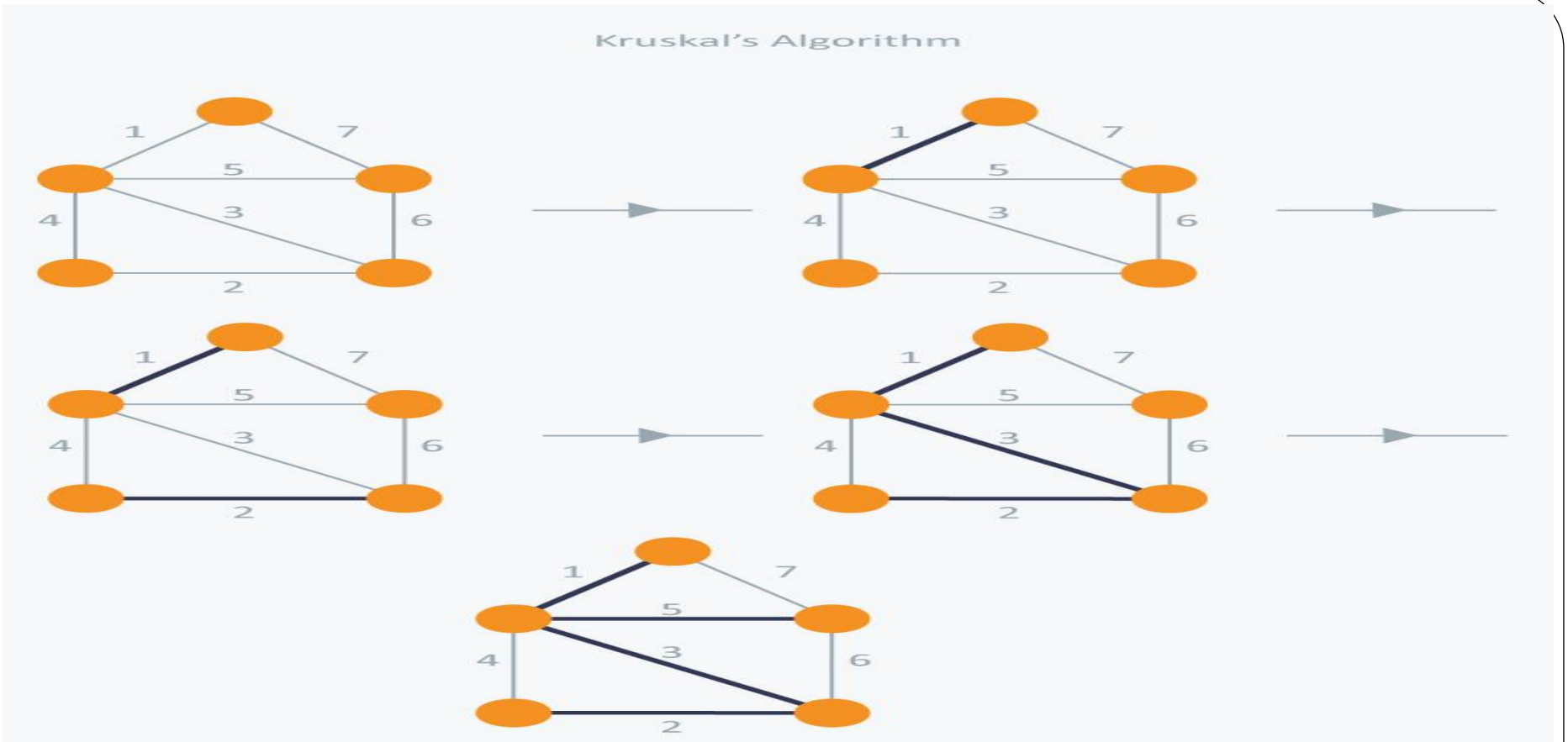
2. Prim's Algorithm

1 Kruskal's Algorithm

- **Kruskal's Algorithm**
- Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.
- **Algorithm Steps:**
- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

Step	Edge Consider	Spanning Tree
1	**Find Spanning tree and Minimal cost using Kruskal's algorithm**	
2	Select Edge (B,C)	
3	Select Edge (C,E)	
4	Select Edge (C,D)	

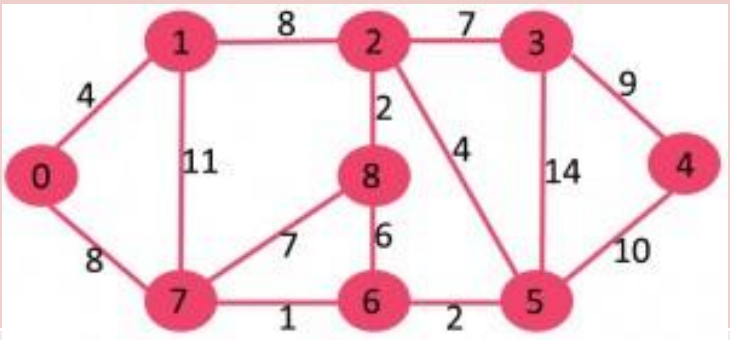
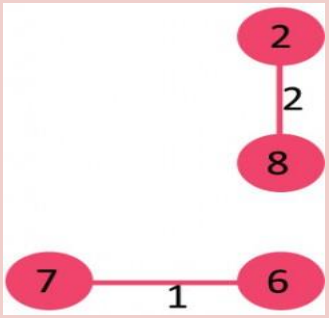
Step	Edge Consider	Description
5	Select Edge (E,F)	 Step: 5
6	Select Edge (A,C)	 Step: 6
7	Minimal Cost = 2+2+3+3+4 =14	
Que.		Find Spanning tree and Minimal cost using kruskal's algorithm

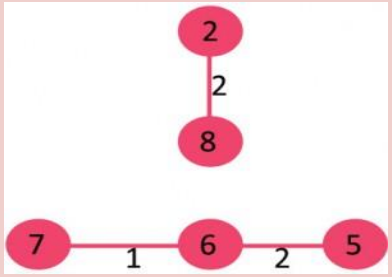
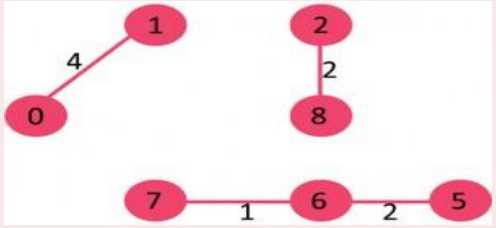
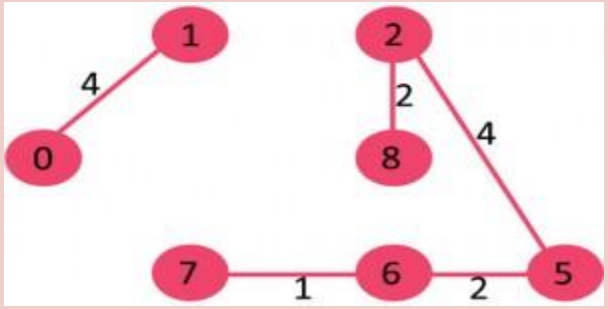
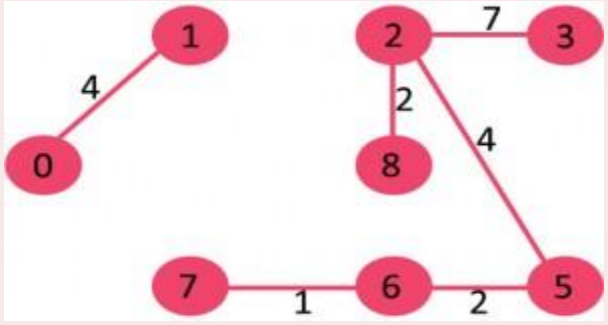


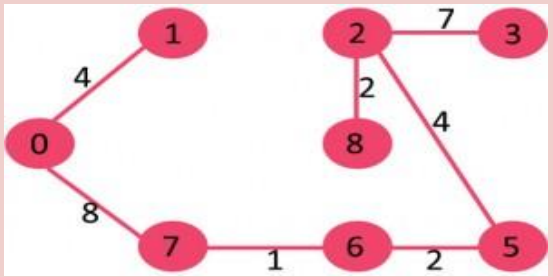
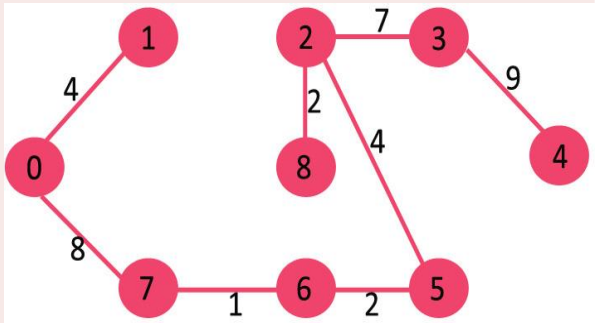
In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

- # Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

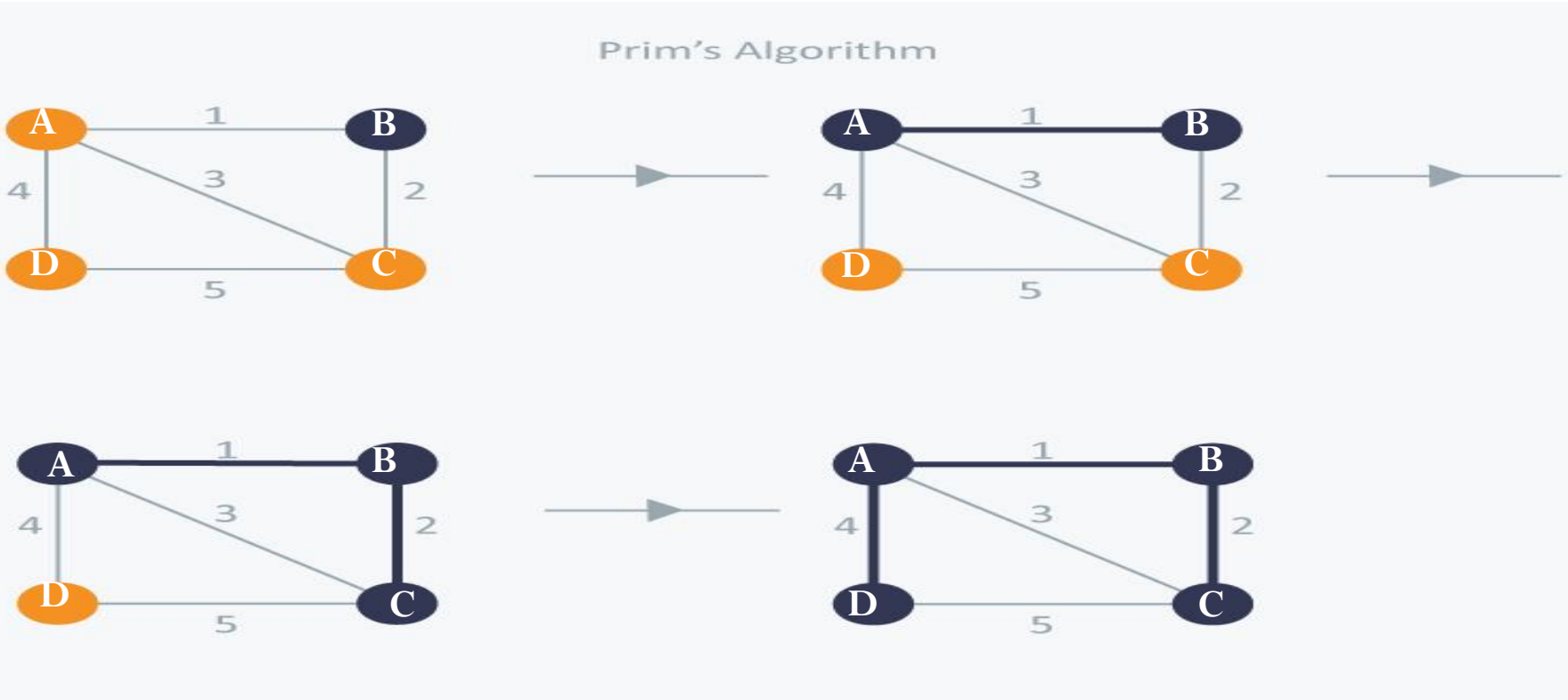
Step	Edge Consider	Description
1		
2		Now pick all edges one by one from sorted list of edges **1.** *Pick edge 7-6:* No cycle is formed, include it.
3		**2.** *Pick edge 8-2:* No cycle is formed, include it.

Step	Edge Consider	Description
4		**3.** *Pick edge 6-5:* No cycle is formed, include it.
5		**4.** *Pick edge 0-1:* No cycle is formed, include it.
6		**5.** *Pick edge 2-5:* No cycle is formed, include it.
7		**6.** *Pick edge 8-6:* Since including this edge cycle formed so discard it. **7.** *Pick edge 2-3:* No cycle is formed, include it.

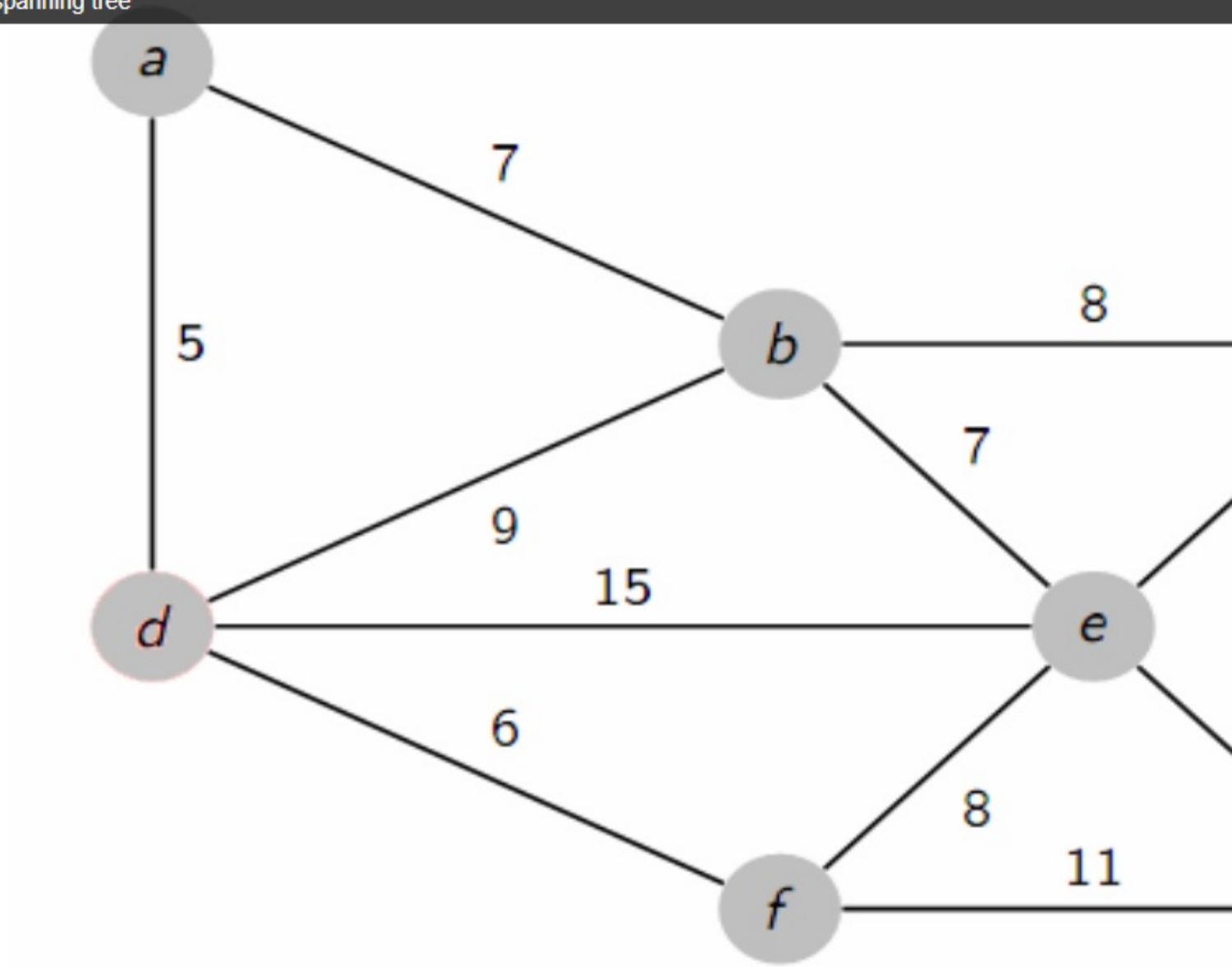
Step	Edge Consider	Description
8		**8.** *Pick edge 7-8:* Since including this edge results in cycle, discard it. **9.** *Pick edge 0-7:* No cycle is formed, include it.
9		**10.** *Pick edge 1-2:* Since including this edge results in cycle, discard it. **11.** *Pick edge 3-4:* No cycle is formed, include it.
10	Since the number of edges included equals (V – 1), the algorithm stops here.	.

Prim's Algorithm

- Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.
- **Algorithm Steps:**
- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.



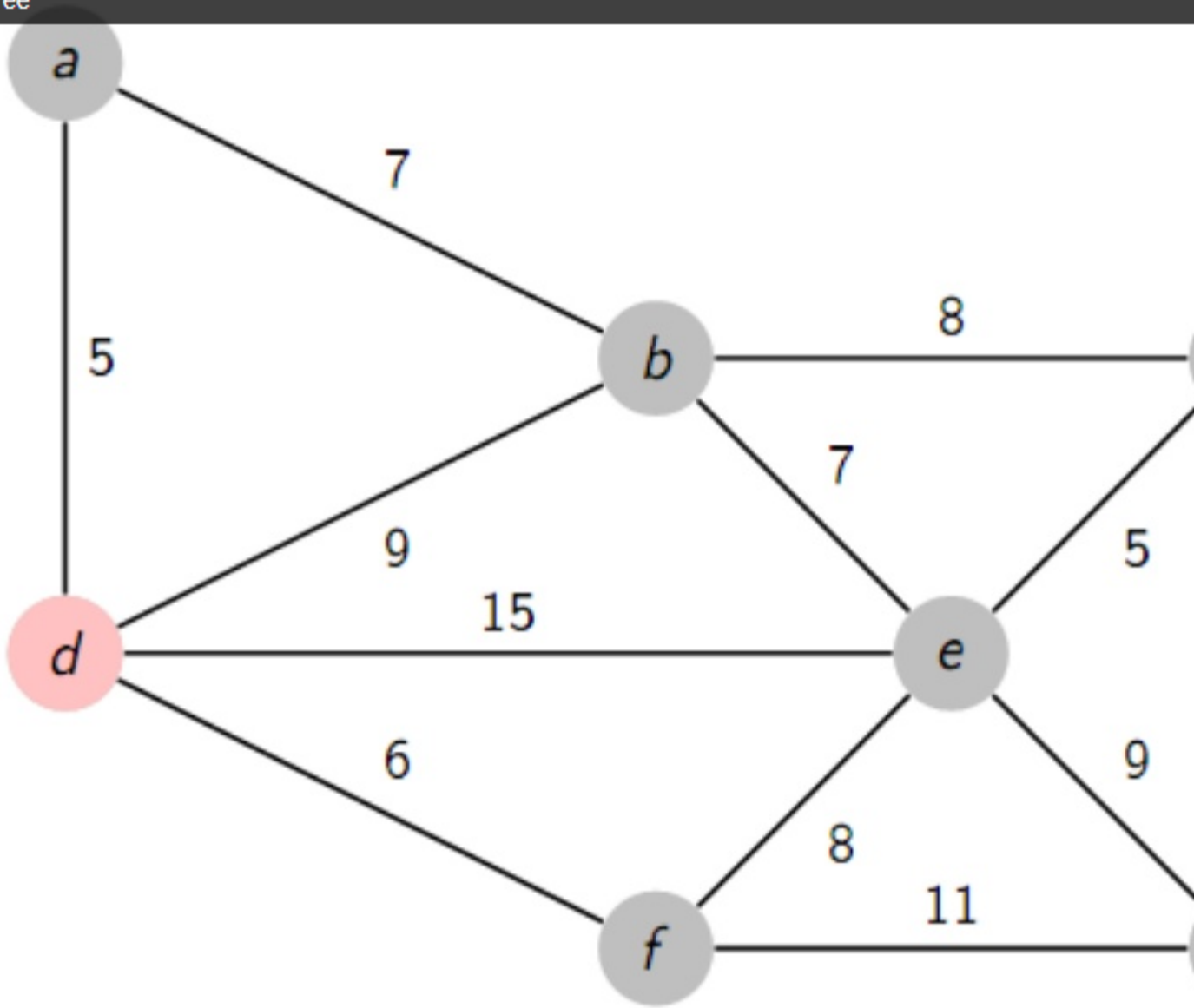
- In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).



Open List: **d**

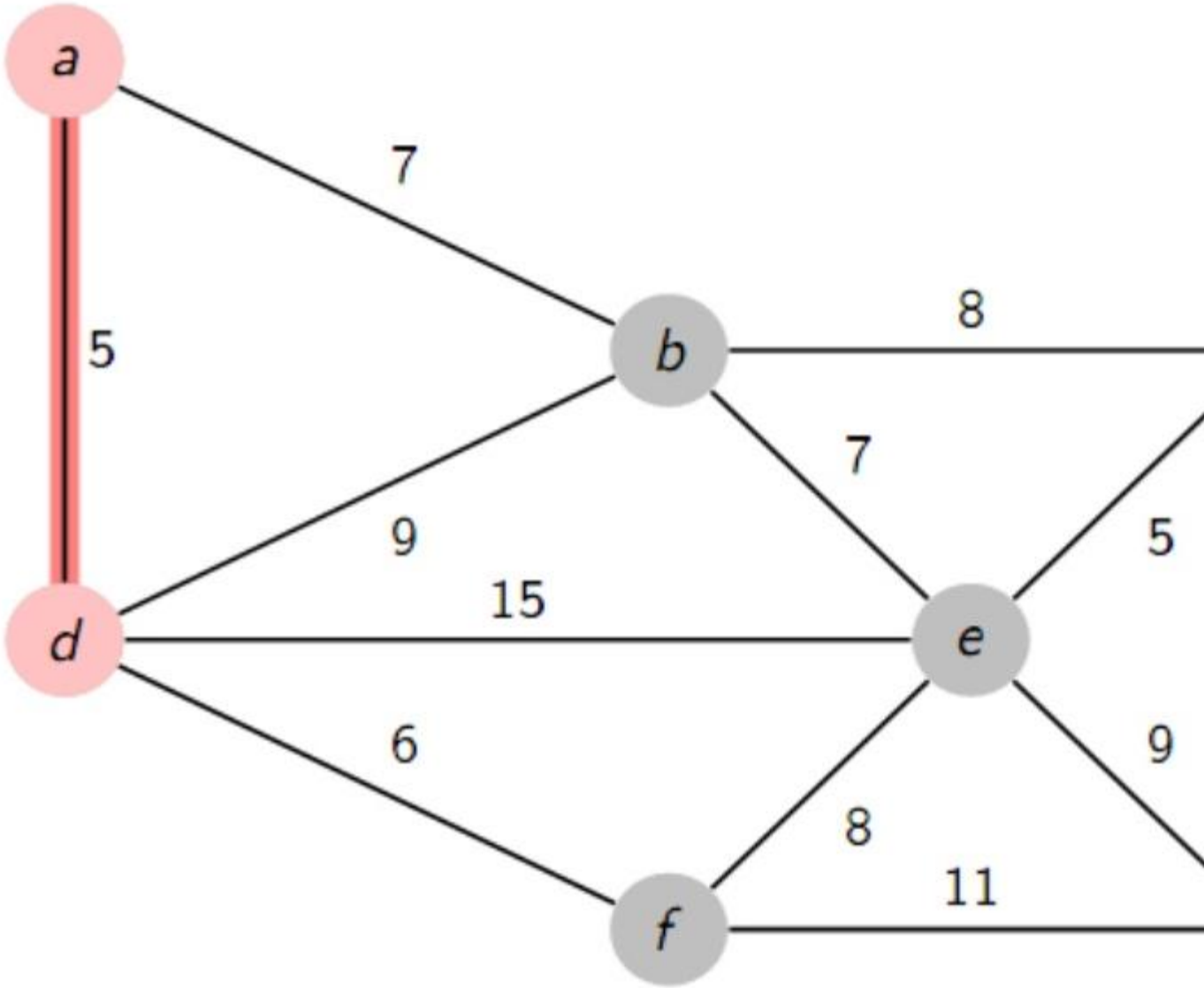
Close List:

Prim's Algorithm on minimum spanning tree



Open List: **a, f, e, b**

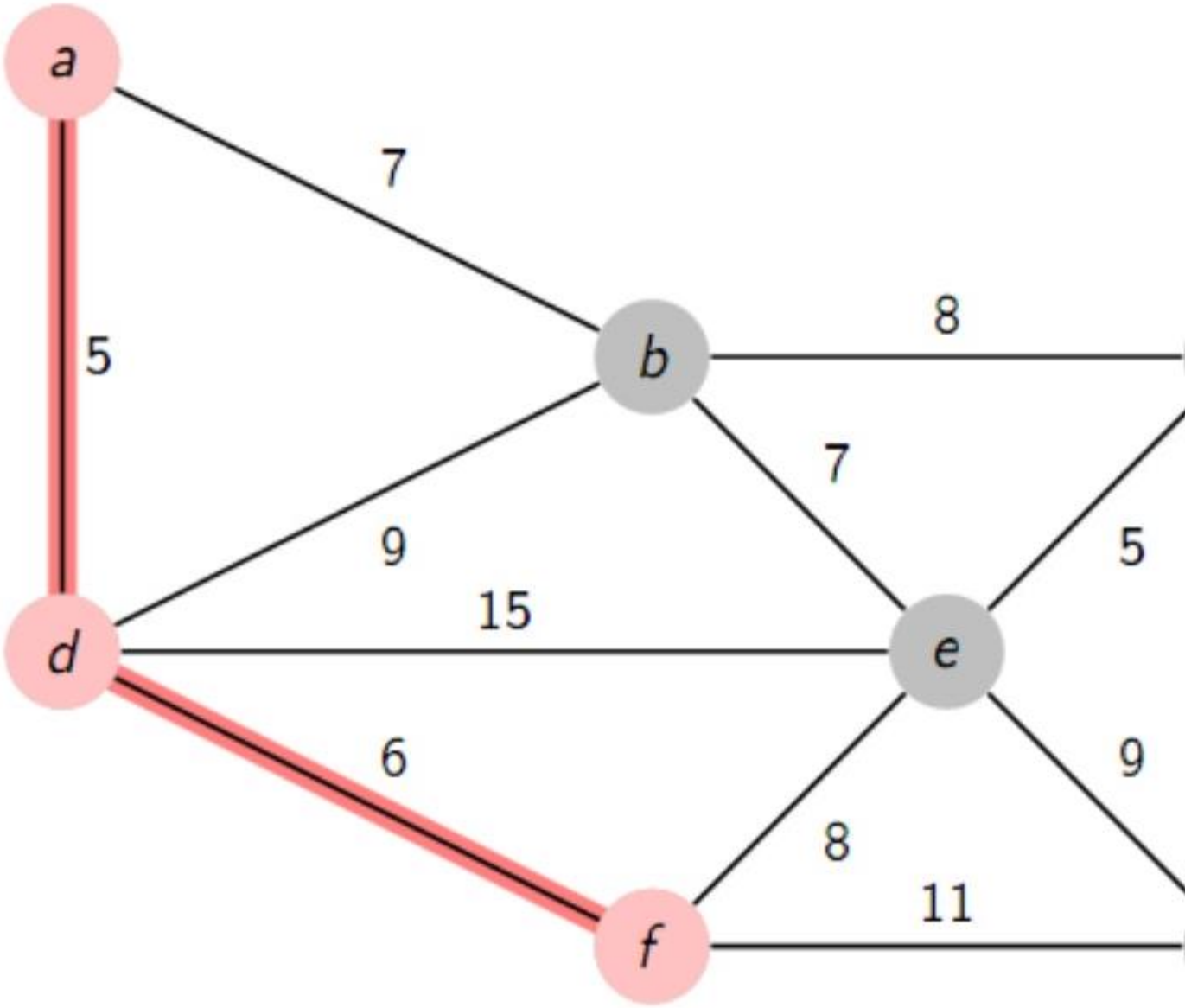
Close List: **d**



Open List: **f, e, b**

Close List: **d, a**

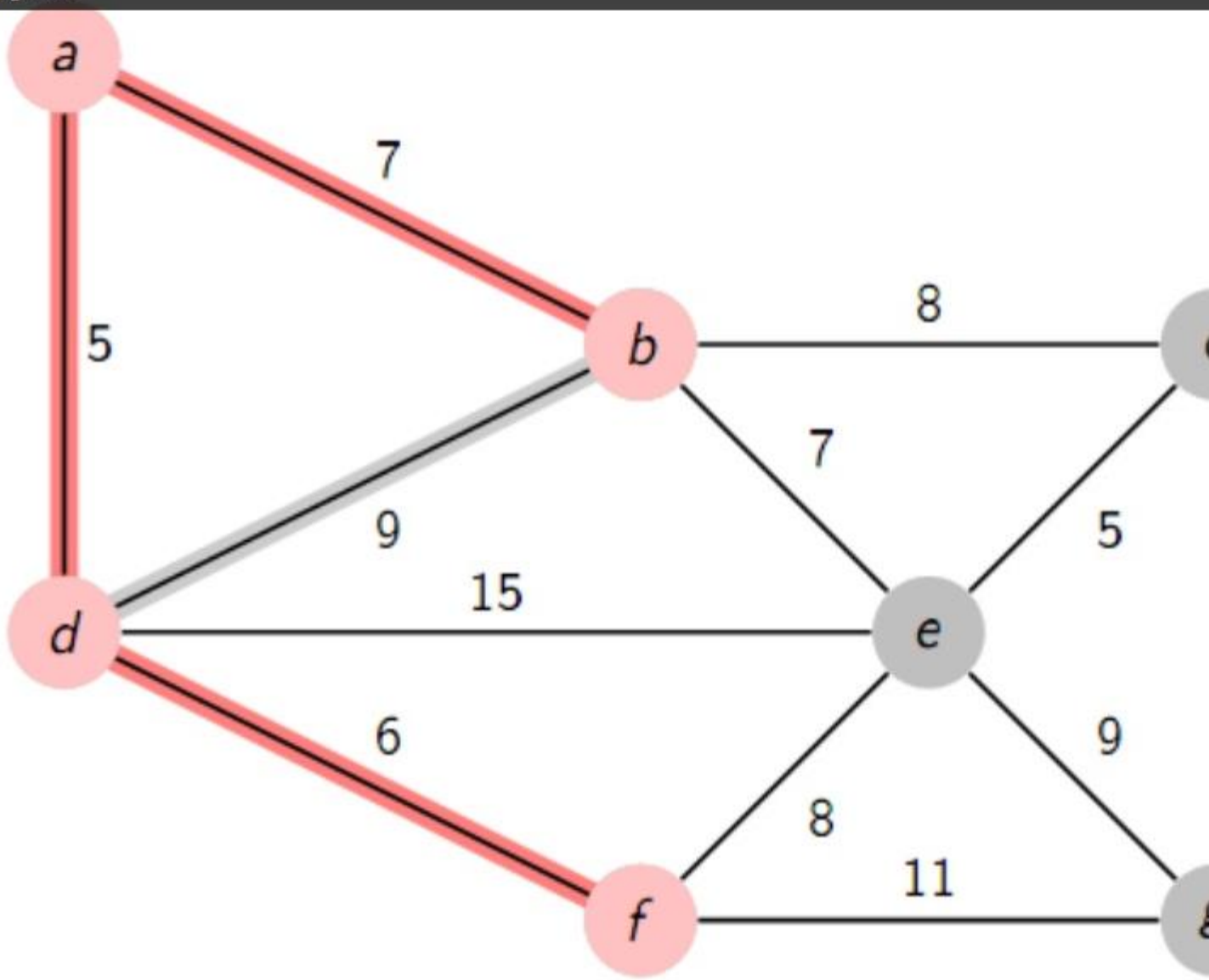
Prim's Algorithm on minimum spanning tree



Open List: **b, e, g**

Close List: **d, a, f**

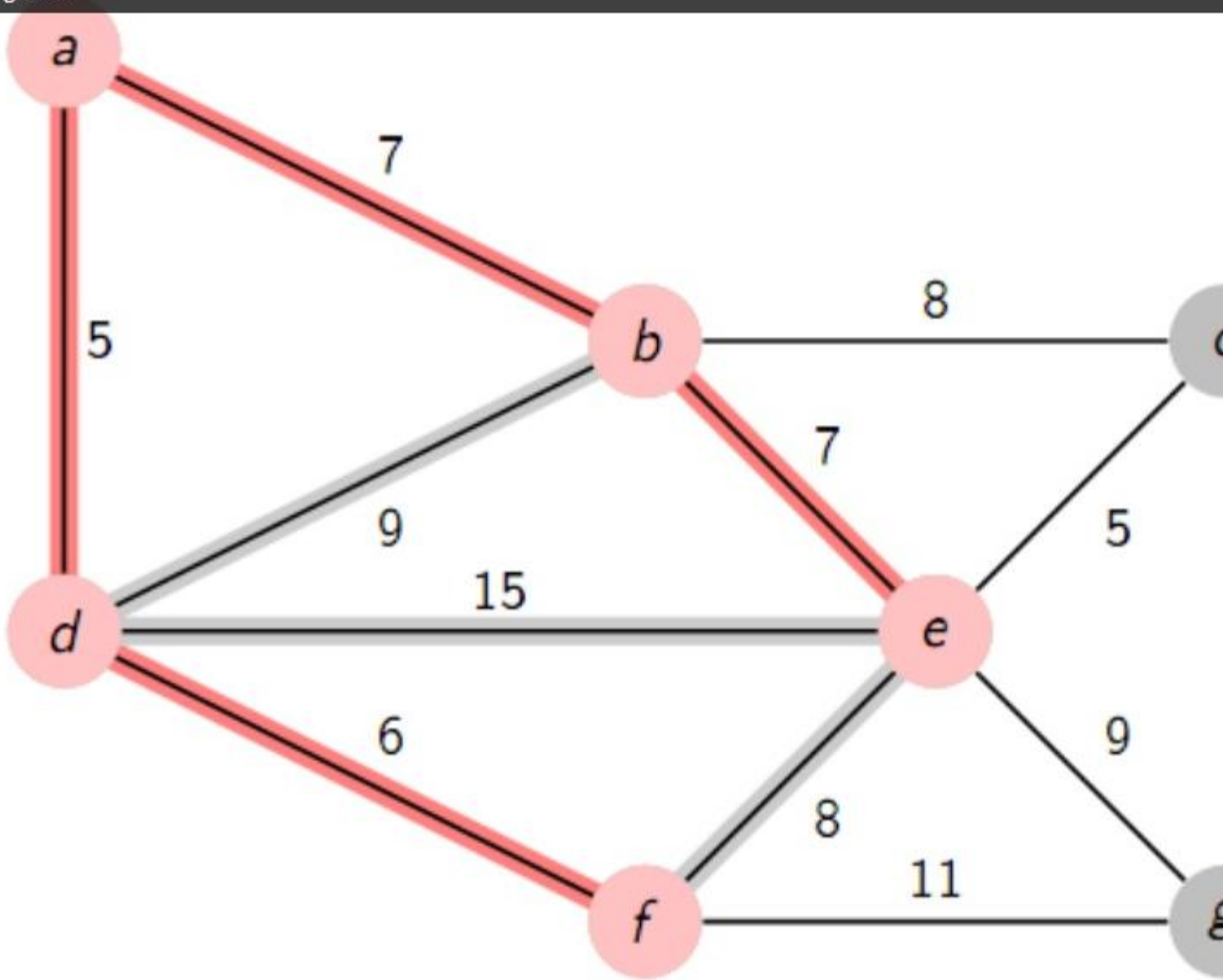
Prim's Algorithm on minimum spanning tree



Open List: **e, g, c**

Close List: **d, a, f, b**

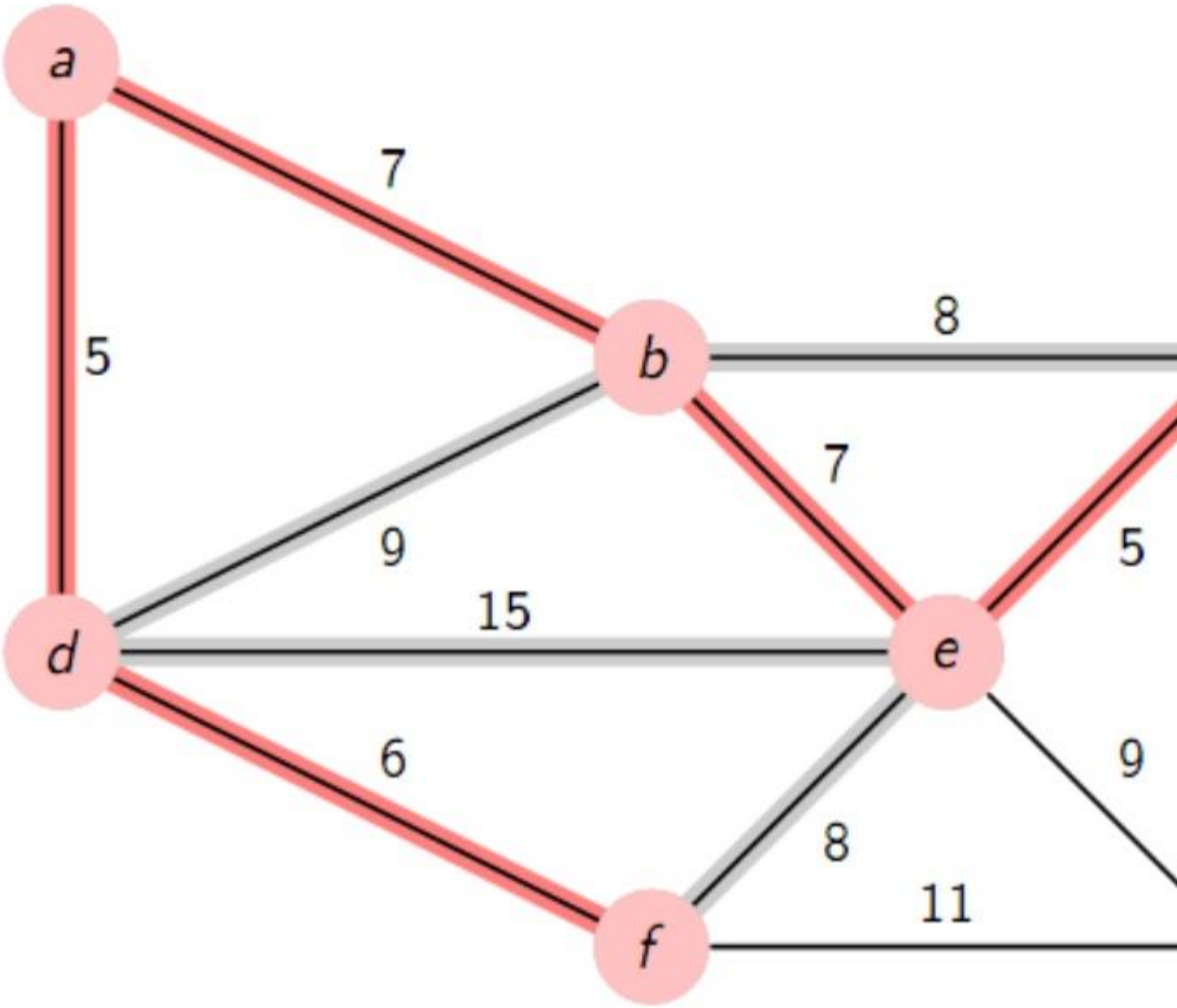
Prim's Algorithm on minimum spanning tree



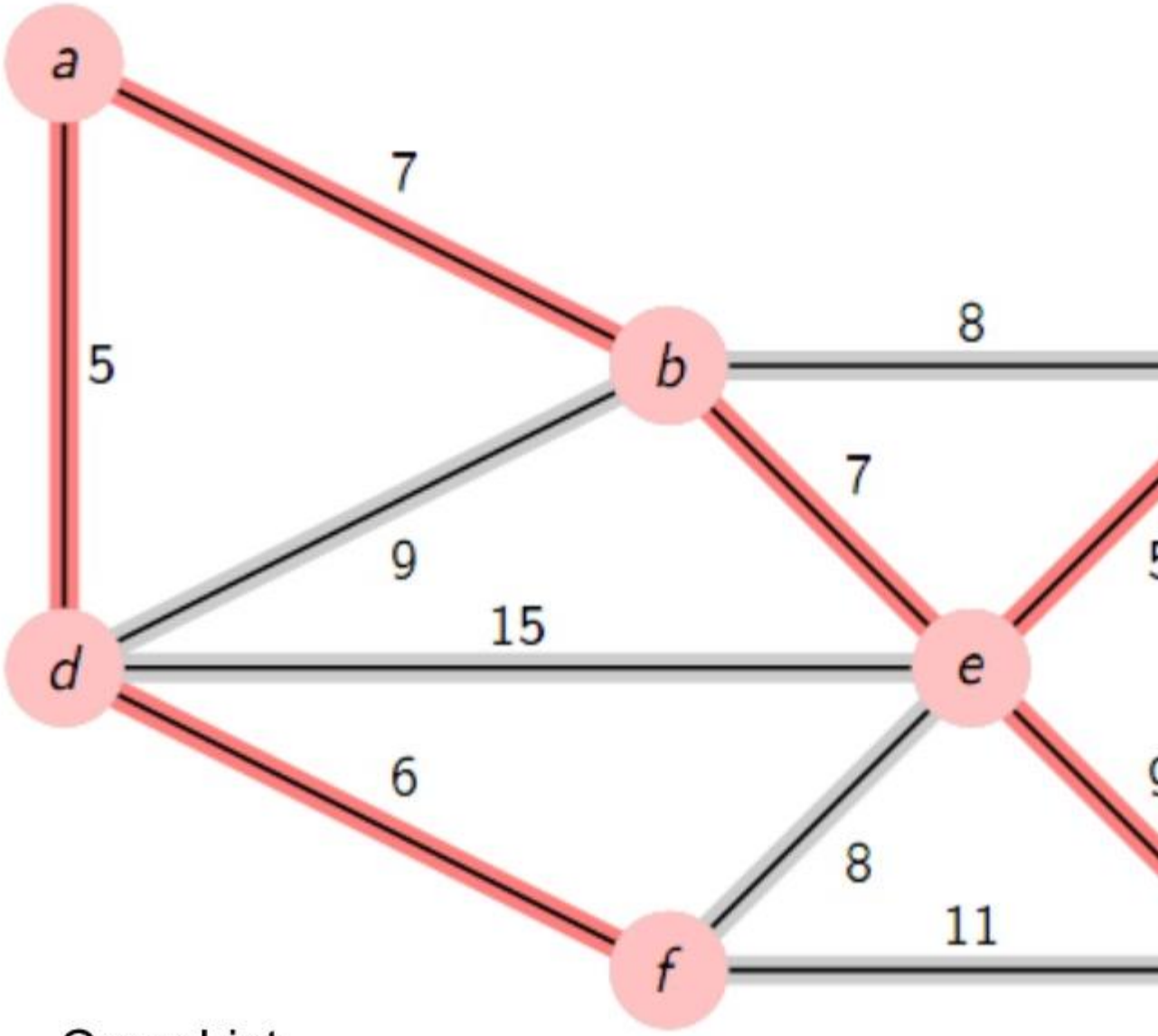
Open List: **c,** g

Close List: **d, a, f, b, e**

Prim's Algorithm on minimum spanning tree



Open List: **g**



Open List:

Close List: d, a, f, b, e, c