Syllabus Of Data Structure

Detailed Contents
Unit -I **Introduction to Data Structures**: Algorithms and Flowcharts, Basics Analysis on Algorithm, Complexity of Algorithm, Introduction and Definition of Data Structure, Classification of Data, Arrays, Various types of Data Structure, Static and Dynamic Memory Allocation, Function, Recursion. **Arrays, Pointers and Strings**: Introduction to Arrays, Definition, One Dimensional Array and Multidimensional Arrays, Pointer, Pointer to Structure, various Programs for Array and Pointer. Strings. Introduction to Strings, Definition, Library Functions of Strings.
Unit-II **Stacks and Queue :-** Introduction to Stack, Definition, Stack Implementation, Operations of Stack, Applications of Stack and Multiple Stacks. Implementation of Multiple Stack Queues, Introduction to Queue, Definition, Queue Implementation, Operations of Queue, Circular Queue, De-queue and Priority Queue.
Unit-III **Linked Lists and Trees** Introduction, Representation and Operations of Linked Lists, Singly Linked List, Doubly Linked List, Circular Linked List, And Circular Doubly Linked List. **Trees** Introduction to Tree, Tree Terminology Binary Tree, Binary Search Tree, Strictly Binary Tree, Complete Binary Tree, Tree Traversal, Threaded Binary Tree, AVL Tree B Tree, B+ Tree.
Unit -IV **Graphs, Searching, Sorting and Hashing Graphs:** Introduction, Representation to Graphs, Graph Traversals Shortest Path Algorithms. **Searching and Sorting:** Searching, Types of Searching, Sorting, Types of sorting like quick sort, bubble sort, merge sort, selection sort. **Hashing:** Hash Function, Types of Hash Functions, Collision, Collision Resolution Technique (CRT), Perfect Hashing

INDEX

UNIT I

❖ Introduction to Data Structures

Basicconceptofdata

Data

Data is a raw and unorganized fact that required to be processed to make it meaningful. Data can be simple at the same time unorganized unless it is organized. Generally, data comprises facts, observations, perceptions numbers, characters, symbols, image, etc. Data is always interpreted, by a human or machine, to derive meaning. So, data is meaningless. Data contains numbers, statements, and characters in a raw form.

Examples of data are weights, prices, costs, numbers of items sold, employee names, product names, addresses, tax codes, registration marks etc

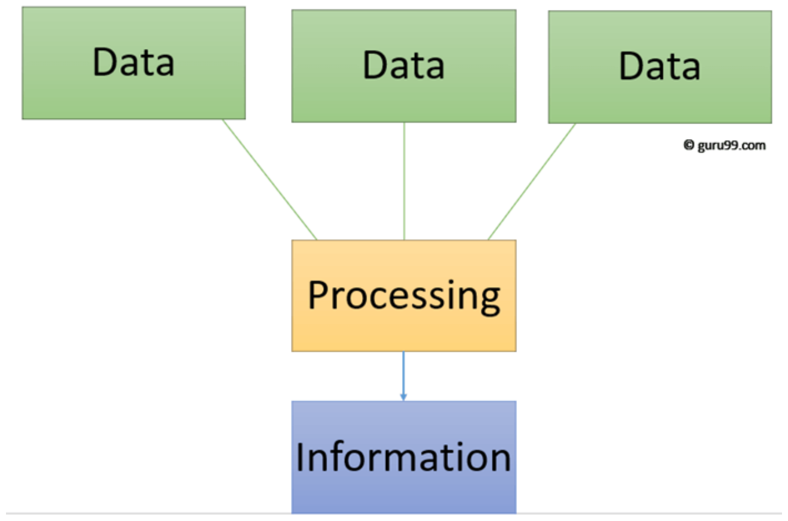
Information

Information is a set of data which is processed in a meaningful way according to the given requirement. Information is processed, structured, or presented in a given context to make it meaningful and useful.

It is processed data which includes data that possess context, relevance, and purpose. It also involves manipulation of raw data.

Information assigns meaning and improves the reliability of the data. It helps to ensure undesirability and reduces uncertainty. So, when the data is transformed into information, it never has any useless details.

Example :- Information is data that has been converted into a more useful or intelligible form.



Difference between Data and Information

Data	**Information**
Data is in the form of numbers, letters, or a set of characters.	Ideas and inferences
It can be structured, tabular data, graph, data tree, etc.	Language, ideas, and thoughts based on the given data.
Data does not have any specific purpose.	It carries meaning that has been assigned by interpreting data.
Data that is collected	Information that is processed.
Data is a single unit and is raw. It alone doesn't have any meaning.	Information is the product and group of data which jointly carry a logical meaning.
It never depends on Information	It depended on Data.
Measured in bits and bytes.	Measured in meaningful units like time, quantity, etc.
It can't be used for decision making	It is widely used for decision making.
It is low-level knowledge.	It is the second level of knowledge.
Data depends upon the sources for collecting data.	Information depends upon data.

❖ Problem Analysis

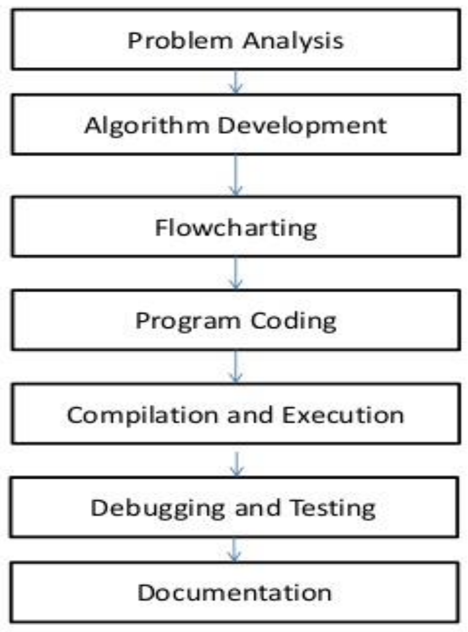


Fig. Steps in problem solving

Ashim Lamichhane 6

Problem Analysis: Identify the issues. Be clear about what the problem is. ...

Understand everyone's interests. ...

List the possible solutions (options) ...

Evaluate the options.

Select an option or options. ...

Document the agreement(s). ...

Agree on contingencies, monitoring, and evaluation.

Algorithm Development

An algorithm in general is a sequence of steps to solve a particular problem. Algorithms are universal. The algorithm you use in C programming language is also the same algorithm you use in every other language. An algorithm produces the same output information given the same input information, and several short algorithms can be combined to perform complex tasks such as writing a computer program.

Flow Chart

A flowchart is a formalized graphic representation of a logic sequence, work or manufacturing process, organization chart, or similar formalized structure. The purpose of a flow chart is to provide people with a common language or reference point when dealing with a project or process. Flowcharts use simple geometric symbols and arrows to define relationships.

Program Coding

A **Programming** (or **coding**) language is a set of syntax rules that define how code should be written and formatted. Thousands of different **programming** languages make it possible for us to create computer **software**, apps and websites.

Compile and Execution

Compilation

First, the source '.java' file is passed through the compiler, which then encodes the source code into a machine independent encoding, known as Bytecode. The content of each class contained in the source file is stored in a separate '.class' file. While converting the source code into the bytecode.

Execution

The class files generated by the compiler are independent of the machine or the OS, which allows them to be run on any system. To run, the main class file (the class that contains the method main) is passed to the JVM, and then goes through three main stages before the final machine code is executed.

Debugging and testing

Testing means verifying correct behavior. Testing can be done at all stages of module development: requirements analysis, interface design, algorithm design, implementation, and integration with other modules. In the following, attention will be directed at implementation testing. Implementation testing is not restricted to execution testing. An implementation can also be tested using correctness proofs, code tracing, and peer reviews, as described below.

Debugging is a cyclic activity involving execution testing and code correction. The testing that is done during debugging has a different aim than final module testing. Final module testing aims to demonstrate correctness, whereas testing during debugging is primarily aimed at locating errors. This difference has a significant effect on the choice of testing strategies.

Documentation

The documentation section contains a set of comment including the name of the program other necessary details. Comments are ignored by compiler and are used to provide documentation to people who reads that code.

❖ Algorithm

An algorithm is defined as a step-by-step procedure or method for solving a problem by a computer in a finite number of steps. Steps of an algorithm definition may include branching or repetition depending upon what problem the algorithm is being developed for. While defining an algorithm steps are written in human understandable language and independent of any programming language. We can implement it in any programming language of our choice.
Besides merely being a finite set of rules which gives a sequence of operations for solving a specific type of problem, a well defined algorithm has five important

Features:

Finiteness. An algorithm must always terminate after a finite number of steps.

Definiteness. Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.

Input. An algorithm has zero or more inputs, i.e, quantities which are given to it initially before the algorithm begins.

Output. An algorithm has one or more outputs i.e, quantities which have a specified relation to the inputs.

Effectiveness. An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.

Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

Input: An algorithm must have 0 or well defined inputs.

Output: An algorithm must have 1 or well defined outputs, and should match with the desired output.

Feasibility: An algorithm must be terminated after the finite number of steps.

Independent: An algorithm must have step-by-step directions which is independent of any programming code.

Unambiguous: An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

Qualities of a good algorithm

Input and output should be defined precisely.

Each steps in algorithm should be clear and unambiguous.

Algorithm should be most effective among many different ways to solve a problem.

An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

Write an algorithm to add two numbers entered by user.

Step 1: Start

```
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Write an algorithm to find the largest among three different numbers entered by user.

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b
           If a>c
              Display a is the largest number.
           Else
              Display c is the largest number.
        Else
           If b>c
              Display b is the largest number.
           Else
              Display c is the greatest number.
   Step 5: Stop
```

Advantages of Algorithms:

- It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
- An algorithm uses a definite procedure.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.
- By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Alogorithms is Time consuming.
- Difficult to show Branching and Looping in Algorithms.
- Big tasks are difficult to put in Algorithms.

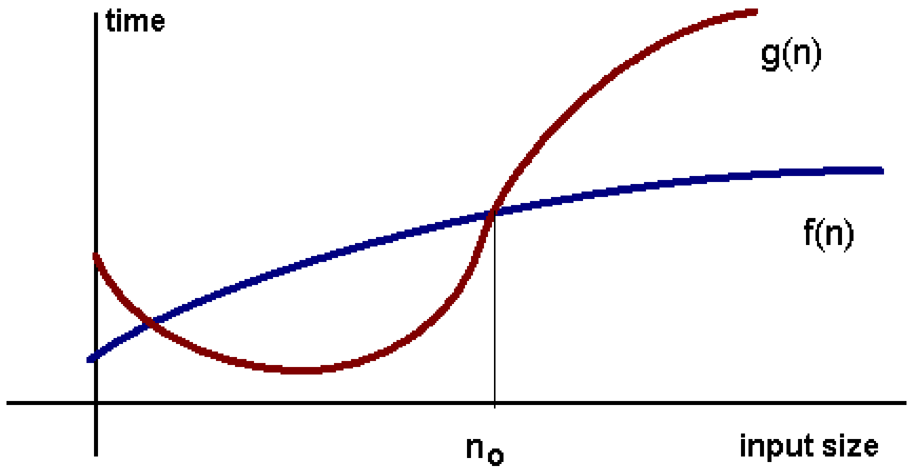
❖ Big O

For any monotonic functions f(n) and g(n) from the positive integers to the positive integers, we say that $f(n) = O(g(n))$ when there exist constants $c > 0$ and $n_0 > 0$ such that

$f(n) \leq c * g(n)$, for all $n \geq n_0$

Intuitively, this means that function f(n) does not grow faster than g(n), or that function g(n) is an upper bound for f(n), for all sufficiently large $n \rightarrow \infty$

Here is a graphic representation of $f(n) = O(g(n))$ relation:



Examples:

1 = O(n)

n = O(n^2)

log(n) = O(n)

Constant Time: O(1)

An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Examples:

array: accessing any element

fixed-size stack: push and pop methods

fixed-size queue: enqueue and dequeue methods

Linear Time: O(n)

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:

array: linear search, traversing, find minimum

ArrayList: contains method

queue: contains method

Logarithmic Time: O(log n)

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size. **Example:**

Binary Search

Recall the "twenty questions" game - the task is to guess the value of a hidden number in an interval. Each time you make a guess, you are told whether your guess iss too high or too low. Twenty questions game imploies a strategy that uses your guess number to halve the interval size. This is an example of the general problem-solving method known as binary search:

locate the element a in a sorted (in ascending order) array by first comparing a with the middle element and then (if they are not equal) dividing the array into two subarrays; if a is less than the middle element you repeat the whole procedure in the left subarray, otherwise - in the right subarray. The procedure repeats until a is found or subarray is a zero dimension.

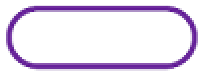
Note, $\log(n) < n$, when $n \to \infty$. Algorithms that run in O(log n) does not use the whole input.

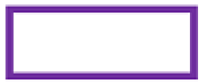
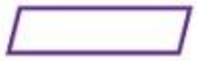
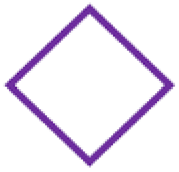
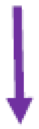
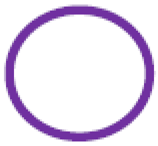
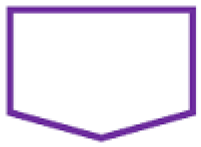
❖ Flowchart

Flowchart is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowchart Symbols

Here is a chart for some of the common symbols used in drawing flowcharts.

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the

		program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

Guidelines for Developing Flowcharts

These are some points to keep in mind while developing a flowchart −

Flowchart can have only one start and one stop symbol

On-page connectors are referenced using numbers

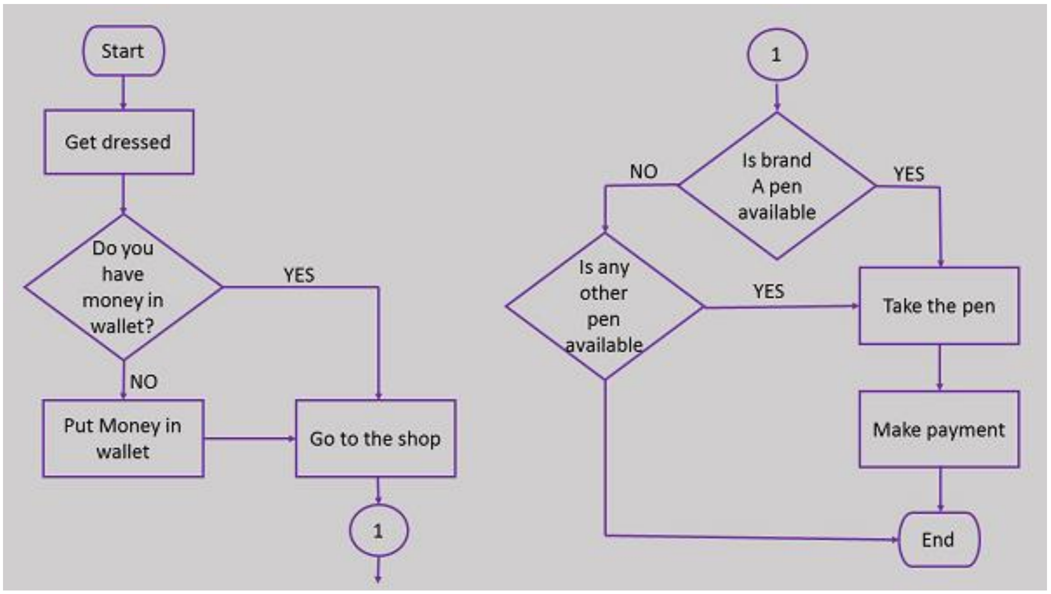
Off-page connectors are referenced using alphabets

General flow of processes is top to bottom or left to right

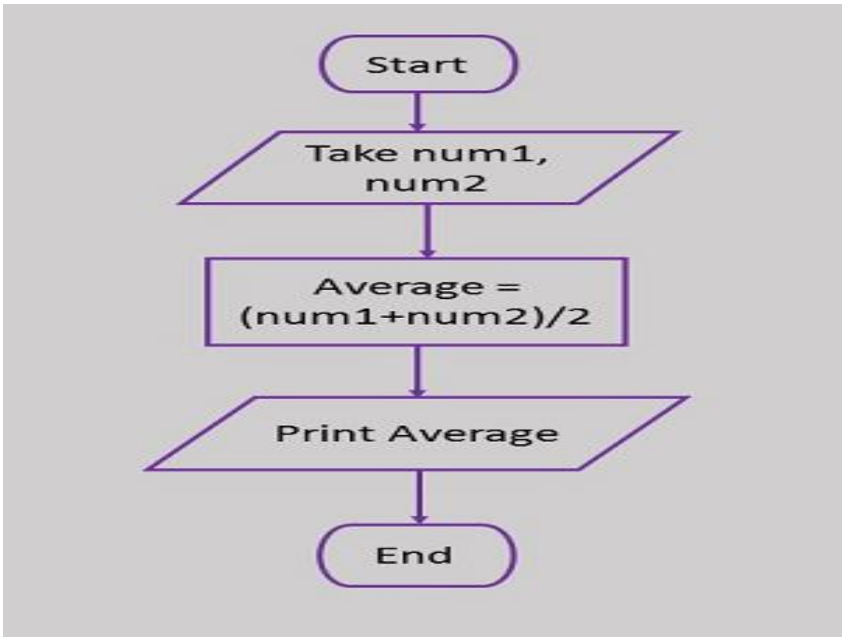
Arrows should not cross each other

Example Flowcharts

Here is the flowchart for going to the market to purchase a pen.



Here is a flowchart to calculate the average of two numbers.



❖ Analysis of Algorithms

The term analysis of algorithms is used to describe approaches to the study of the performance of algorithms. In this course we will perform the following types of analysis:

the worst-case runtime complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size a.

the best-case runtime complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size a.

the average case runtime complexity of the algorithm is the function defined by an average number of steps taken on any instance of size a.

the amortized runtime complexity of the algorithm is the function defined by a sequence of operations applied to the input of size a and averaged over time.

Example. Let us consider an algorithm of sequential searching in an array.of size n.

Its worst-case runtime complexity is O(n)
Its best-case runtime complexity is O(1)
Its average case runtime complexity is O(n/2)=O(n)

Algorithm Complexity

Algorithm Complexity are two types

1) **Time complexity** of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.

Three cases are consider while calculating Time complexity

Worst case: An algorithm takes more time to execution is called the worst case, it is mostly in cases of loops where execution time is depends upon the number of input size.

Average case: An algorithm takes time between the best and worst case to execution is called the Average case.

Best case: An algorithm takes expected to execution is called the best case.

2) Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

Variables (This include the constant values, temporary values)

Program Instruction

Execution

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

Sometime Auxiliary Space is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = Auxiliary Space + Input space

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

Instruction Space

It's the amount of memory used to save the compiled version of instructions.

Environmental Stack

Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the function A().

Data Space

Amount of space used by the variables and constants.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of data type variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

❖ Data Structure and Types

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer

Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vitle role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures.
For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very

efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Characteristics of data structures

Data structures are often classified by their characteristics. Possible characteristics are:

Linear or non-linear: This characteristic describes whether the data items are arranged in chronological sequence, such as with an array, or in an unordered sequence, such as with a graph.

Homogeneous or non-homogeneous: This characteristic describes whether all data items in a given repository are of the same type or of various types.

Static or dynamic: This characteristic describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time. Dynamic data structures have sizes, structures and memory locations that can shrink or expand depending on the use.

❖ A DEFINITION OF DATA CLASSIFICATION

Data classification is broadly defined as the process of organizing data by relevant categories so that it may be used and protected more efficiently. On a basic level, the classification process makes data easier to locate and retrieve. Data classification is of particular importance when it comes to risk management, compliance, and data security.

Data classification involves tagging data to make it easily searchable and traceable. It also eliminates multiple duplications of data, which can reduce storage and backup costs while speeding up the search process. Though the classification process may sound highly technical, it is a topic that should be understood by your organization's leadership.

REASONS FOR DATA CLASSIFICATION

Data classification has improved significantly over time. Today, the technology is used for a variety of purposes, often in support of data security initiatives. But data may be classified for a number of reasons, including ease of access, **maintaining regulatory compliance** and to meet various other business or personal objectives. In some cases, data classification is a regulatory requirement, as data must be searchable and retrievable within specified timeframes. For the purposes of data security, data classification is a useful tactic that facilitates proper security responses based on the type of data being retrieved, transmitted, or copied.

❖ TYPES OF DATA CLASSIFICATION

Data classification often involves a multitude of tags and labels that define the type of data, its confidentiality, and its integrity. Availability may also be taken into consideration in data classification processes. Data's level of sensitivity is often classified based on varying levels of importance or confidentiality, which then correlates to the security measures put in place to protect each classification level.

There are **three main types of data classification** that are considered industry standards:

Content-based classification inspects and interprets files looking for sensitive information

Context-based classification looks at application, location, or creator among other variables as indirect indicators of sensitive information

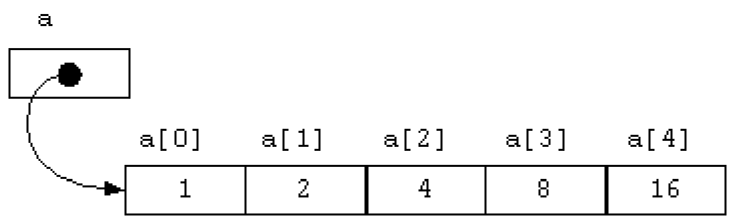
User-based classification depends on a manual, end-user selection of each document. User-based classification relies on user knowledge and discretion at creation, edit, review, or dissemination to flag sensitive documents.

Content-, context-, and user-based approaches can be both right or wrong depending on the business need and data type.

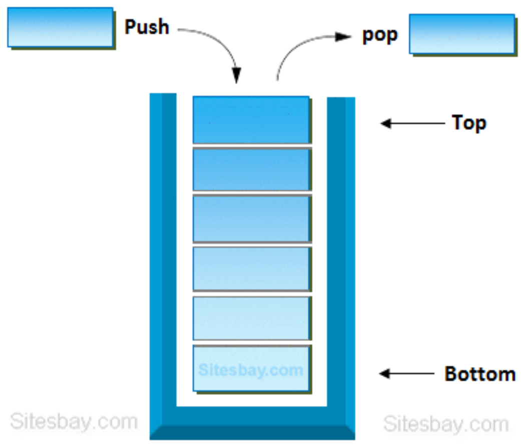
❖ Types of data structures

Arrays- An array stores a collection of items at adjoining memory locations. Items that are the same type get stored together so that the position of each element can be calculated or retrieved easily. Arrays can be fixed or flexible in length.

Example



Stacks- A stack stores a collection of items in the linear order that operations are applied. This order could be last in first out (LIFO).**For example** Stack of coins, stack of plates ,stack of books.



Mainly the following three basic operations are performed in the stack:

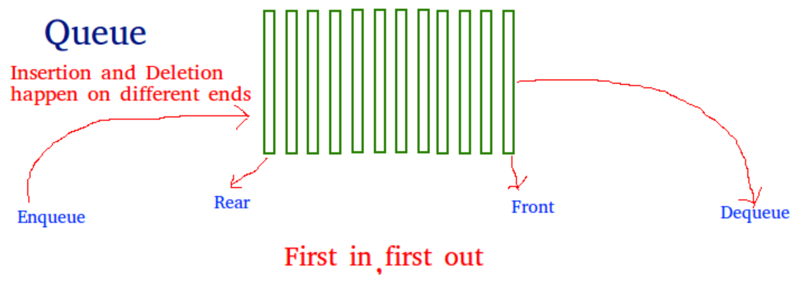
Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek or Top: Returns top element of stack.

Empty: Returns true if stack is empty, else false.

Queues- A queue stores a collection of items similar to a stack; however, the operation order can only be first in first out.



Operations on Queue:
Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

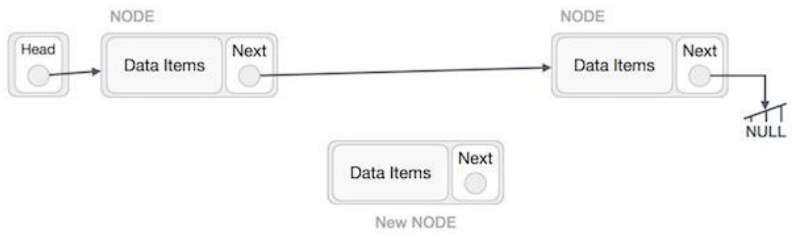
Front: Get the front item from queue.

Rear: Get the last item from queue.

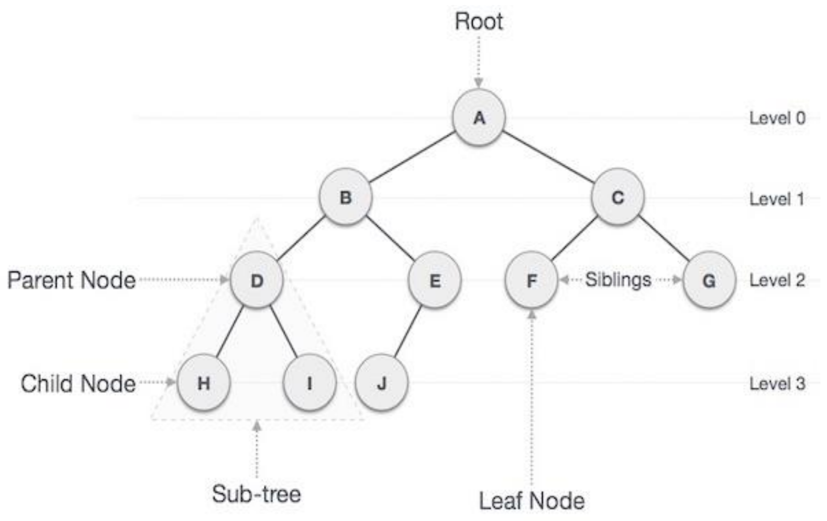
Linked lists- A linked list stores a collection of items in a linear order. Each element, or node, in a linked list contains a data item as well as a reference, or link, to the next item in the list.

Basic Operations.

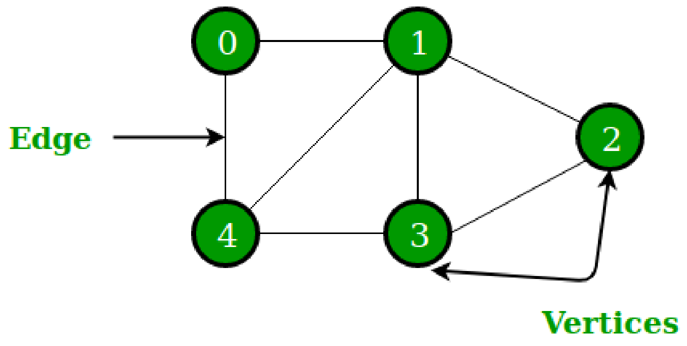
- Insertion Operation.
- Deletion Operation.
- Reverse Operation.



Trees- A tree stores a collection of items in an abstract, hierarchical way. Each node is linked to other nodes and can have multiple sub-values, also known as children.



Graphs- A graph stores a collection of items in a non-linear fashion. Graphs are made up of a finite set of nodes, also known as vertices, and lines that connect them, also known as edges. These are useful for representing real-life systems such as computer networks



Blocks :- In computing (specifically data transmission and data storage), a block, sometimes called a physical record, is a sequence of bytes or bits, usually containing some whole number of records, having a maximum length, a block size. Data thus structured are said to be blocked.

The process of putting data into blocks is called *blocking*.

deblockin*g* is the process of extracting data from blocks.

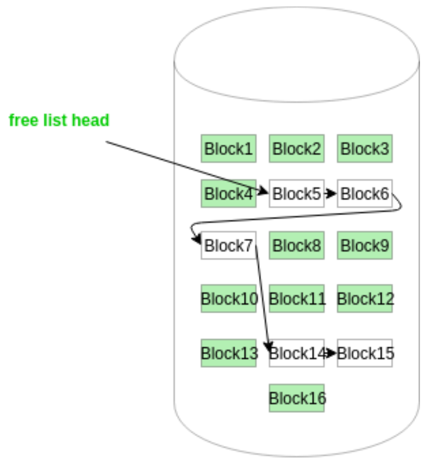
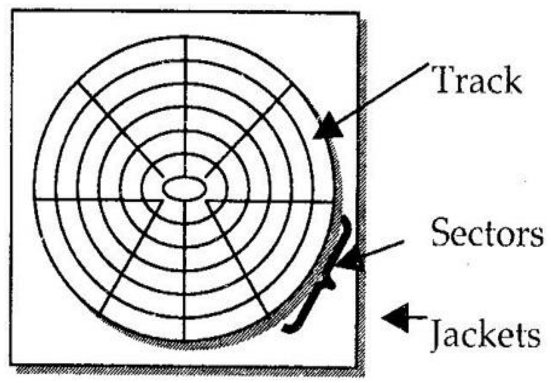


Figure - 2

The sector is the minimum storage unit of a hard drive. Most disk partitioning schemes are designed to have files occupy an integral number of sectors regardless of the file's actual size. The sector means a portion of a disk between a center, two radii and a corresponding arc (see Figure 1, item B), which is shaped like a slice of a pie. Thus, the disk sector refers to the intersection of a track and geometrical sector.

Tracks and Spots
A disk's surface is divided into concentric tracks (circles within circles), and the thinner the tracks, the more storage. The data bits are recorded as magnetic spots on the tracks, and the smaller the spot, the greater the storage.



❖ **What is static memory allocation and dynamic memory allocation?**

Static Memory Allocation: Memory is allocated for the declared variable by the compiler. The address can be obtained by using 'address of' operator and can be assigned to a pointer. The memory is allocated during compile time. Since most of the declared variables have static memory, this kind of assigning the address of a variable to a pointer is known as static memory allocation.

Dynamic Memory Allocation: Allocation of memory at the time of execution (run time) is known as dynamic memory allocation. The functions calloc() and malloc() support allocating of dynamic memory. Dynamic allocation of memory space is done by using these functions when value is returned by functions and assigned to pointer variables.

Reasons and Advantage of allocating memory dynamically:

When we do not know how much amount of memory would be needed for the program beforehand.

When we want data structures without any upper limit of memory space.

When you want to use your memory space more efficiently. **Example***:* If you have allocated memory space for a 1D array as array[20] and you end up using only 10 memory spaces then the remaining 10 memory spaces would be wasted and this wasted memory cannot even be utilized by other program variables.

Dynamically created lists insertions and deletions can be done very easily just by the manipulation of addresses whereas in case of statically allocated memory insertions and deletions lead to more movements and wastage of memory.

When you want you to use the concept of structures and linked list in programming, dynamic memory allocation is a must.

❖ What is recursion?

In simple words, recursion is a problem solving, and in some cases, a programming technique that has a very special and exclusive property. In recursion, a function or method has the ability of calling itself to solve the problem. The process of recursion involves solving a problem by turning it into smaller varieties of itself.

The process in which a function calls itself could happen directly as well as indirectly. This difference in call gives rise to different types of recursion, which we will talk about a little later. Some of the problems that can be solved using recursion include DFS of Graph, Towers of Hanoi, Different Types of Tree Traversals, and others.

Examples of Recursion

- Factorial of a positive integer
- Fibonacci series

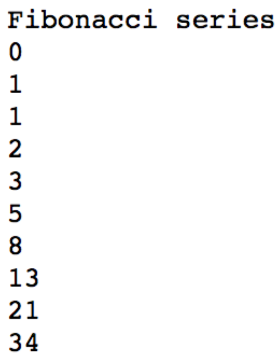
- Greatest common divisor
- Tower of Hanoi

1. **Factorial of a positive integer**:- Factorial is a mathematical term. Factorial of a number,say n, is equal to the product of all integers from 1 to n. Factorial of n is dentoed

By

n! = 1*2*3…..*n.

2**. Fibonacci series**:- Another well known mathematical recursive function is one that computers the fiboncci numbers.



```
Fibonacci series
0
1
1
2
3
5
8
13
21
34
```

Greatest common divisor:- In mathematics, the greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For example, the gcd of 8 and 12 is 4.

Tower of Hanoi:-

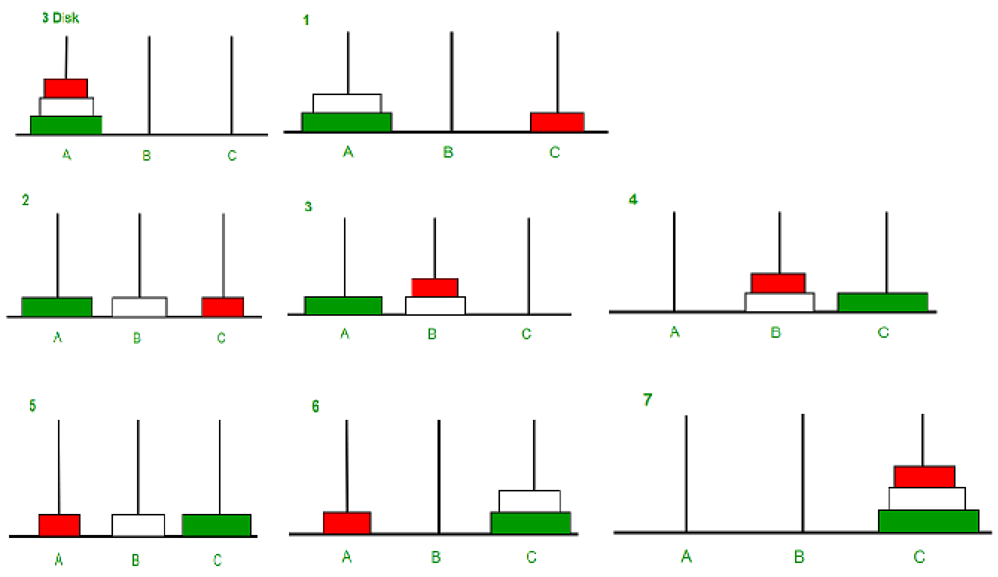
The Tower of Hanoi (also called the Tower of Brahma or Lucas' Tower[1] and sometimes pluralized as Towers) is a mathematical game or puzzle. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

Only one disk can be moved at a time.

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.

No larger disk may be placed on top of a smaller disk.



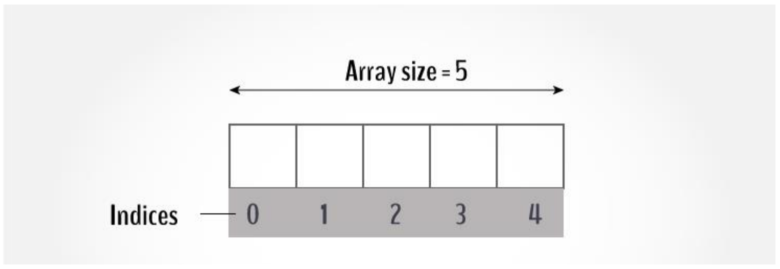
❖ Array

Definition

Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.

Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.

Array is the simplest data structure where each data element can be randomly accessed by using its index number.



Advantages of Array

Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.

Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.

Any element in the array can be directly accessed by using the index.

There are two types of Arrays

- One Dimensional Arrays
- Two Dimensional Arrays

One Dimensional Arrays

A one-dimensional array is one in which only one subscript specification is needed to specify a particular element of the array.

A one-dimensional array is a list of related variables. Such lists are common in programming.

One-dimensional array can be declared as follows :

Data_type var_name[Expression];

Initializing One-Dimensional Array

ANSI C allows automatic array variables to be initialized in declaration by constant initializers as we have seen we can do for scalar variables.

These initializing expressions must be constant value; expressions with identifiers or function calls may not be used in the initializers.

The initializers are specified within braces and separated by commas.

```
int ex[5] = { 10, 5, 15, 20, 25} ;
```

```
char word[10] = { 'h', 'e', 'l', 'l', 'o' } ;
```

Two Dimensional Array

Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts.

Two dimensional array inwhich elements are stored column by column is called as column major matrix.

Two dimensional array in which elements are stored row by row is called as row majo rmatrix.

First subscript denotes number of rows and second subscript denotes the number of columns.

The simplest form of the Multi Dimensionl Array is the Two Dimensionl Array. A Multi Dimensionl Array is essence a list of One Dimensionl Arrays.

Two dimensional arrays can be declared as follows :

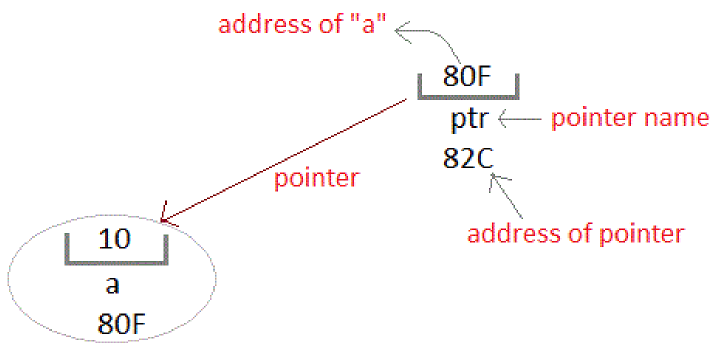
```
int int_array[10] ;         // A normal one dimensional array
int int_array2d[10][10] ;   // A two dimensional array
```

❖ Pointer

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

- Traversing String
- Lookup Tables
- Control Tables

- Tree Structures



NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
#include <stdio.h>


int main () {


  int  *ptr = NULL;


  printf("The value of ptr is : %x\n", ptr  );


  return 0;
}
```

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

```
The value of ptr is 0
```

Pointer Details

Pointer arithmetic: There are four arithmetic operators that can be used in pointers: ++, --, +, -

Array of pointers: You can define arrays to hold a number of pointers.

Pointer to pointer: C allows you to have pointer on a pointer and so on.

Passing pointers to functions in C: Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.

Return pointer from functions in C: C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.

❖ Pointer to structure.

```
#include<stdio.h>


struct dog

{

   char name[10];

   char breed[10];

   int age;

   char color[10];

};


int main()
```

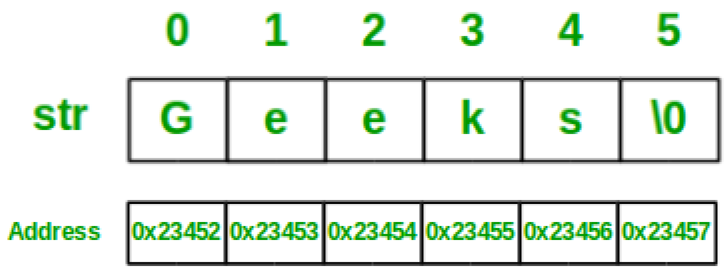
```
{
    struct dog my_dog = {"tyke", "Bulldog", 5, "white"};
    struct dog *ptr_dog;
    ptr_dog = &my_dog;

    printf("Dog's name: %s\n", ptr_dog->name);
    printf("Dog's breed: %s\n", ptr_dog->breed);
    printf("Dog's age: %d\n", ptr_dog->age);
    printf("Dog's color: %s\n", ptr_dog->color);

    // changing the name of dog from tyke to jack
    strcpy(ptr_dog->name, "jack");

    // increasing age of dog by 1 year
    ptr_dog->age++;

    printf("Dog's new name is: %s\n", ptr_dog->name);
    printf("Dog's age is: %d\n", ptr_dog->age);

    // signal to operating system program ran fine
    return 0;
}
```

❖ String

Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character '\0'.

Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string in **C programming** language.

char str_name[size];



❖ Library Functions of Strings

The nine most commonly used functions in the string library are:

strcat - concatenate two strings.

strchr - string scanning operation.

strcmp - compare two strings.

strcpy - copy a string.

strlen - get string length.

strncat - concatenate one string with part of another.

strncmp - compare parts of two strings

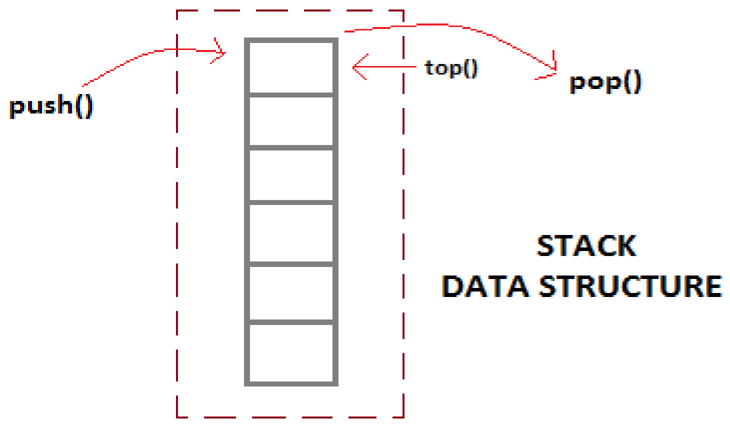
UNIT –II

❖ Stack And Queue

1)**Stacks**: Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack, just like a Stack of coins, books etc.

Basic features of Stack

- Stack is an ordered list of similar data type.
- Stack is a LIFO (Last in First out) structure or we can say FILO (First in Last out).
- push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called Top.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.



❖ Operations of Stack

Mainly the following three basic operations are performed in the stack:

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek or Top: Returns top element of stack.

isEmpty: Returns true if stack is empty, else false.

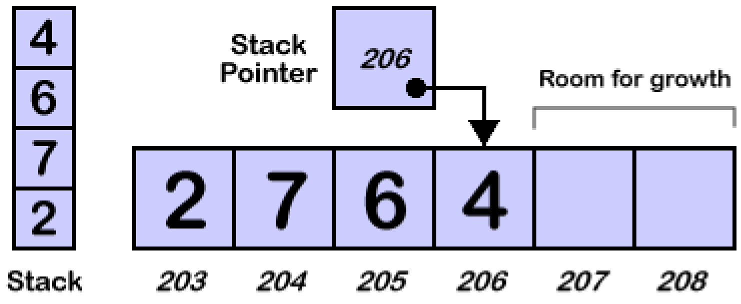
Static Representation of Stack in Memory

There are two ways

- Array
- Linked List

Array (Static) Representation of Stack in Memory

Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Unless otherwise stated or implied, each of our stacks will be maintained by a linear array STACK;a pointer variable TOP, which contains the location of the top element of the stack; and a
variable MAXSTK which gives the maximum number of elements that can be held by the stack. The condition TOP = 0 or TOP = NULL will indicate that the stack is empty.

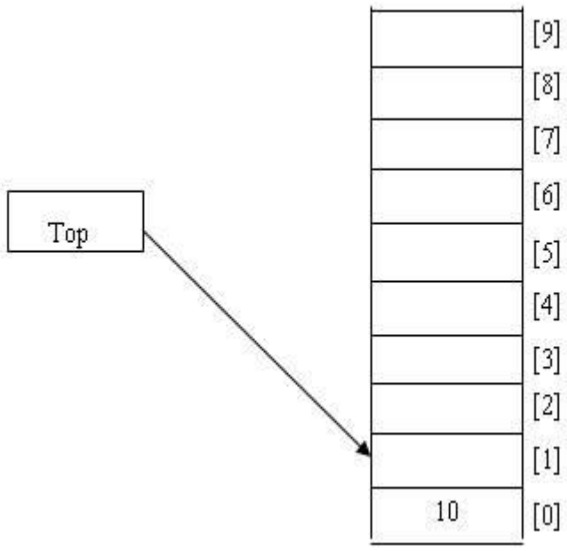


1)Push:

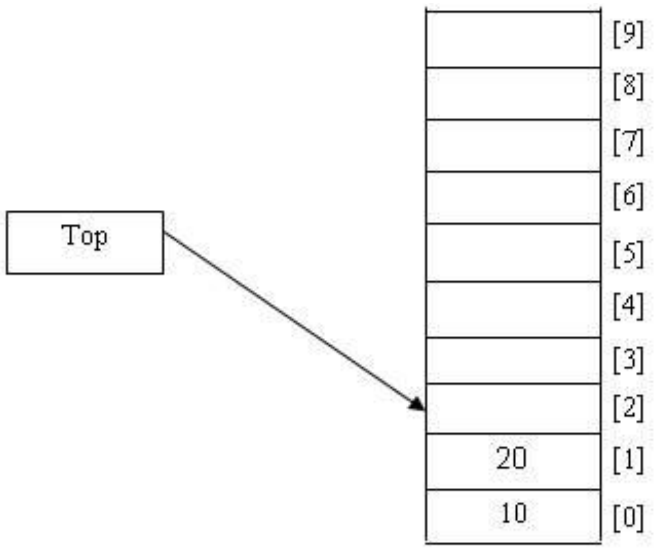
The first thing that needs to be performed is a push operation. The diagram shows the changes to be made if the value 10 is pushed onto the stack:

After the push operation, the array holds the pushed value in the location that used to be indexed by top, and top has now moved on to point to the next slot in the array, ready for the next push. The next step is to perform a second push. The second value to be pushed in this **example is 20**:

Before:



After:



Algorithm For Push Operation

Step 1 − Checks if the stack is full.

Step 2 − If the stack is full, produces an error and exit.

Step 3 − If the stack is not full, increments top to point next empty space.

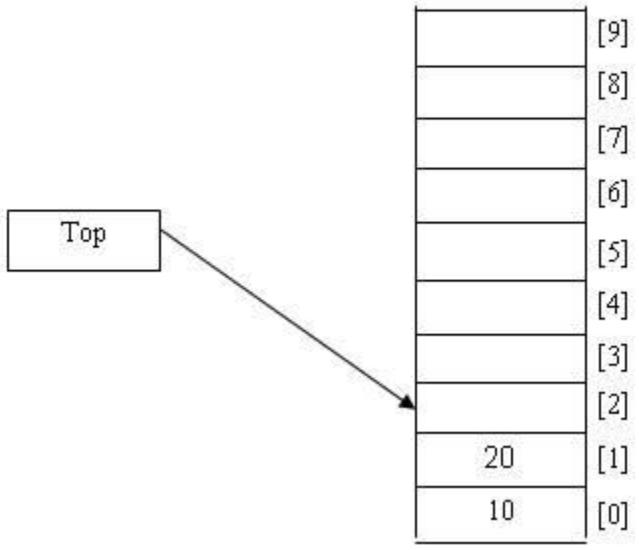
Step 4 − Adds data element to the stack location, where top is pointing.

Step 5 − Returns success.

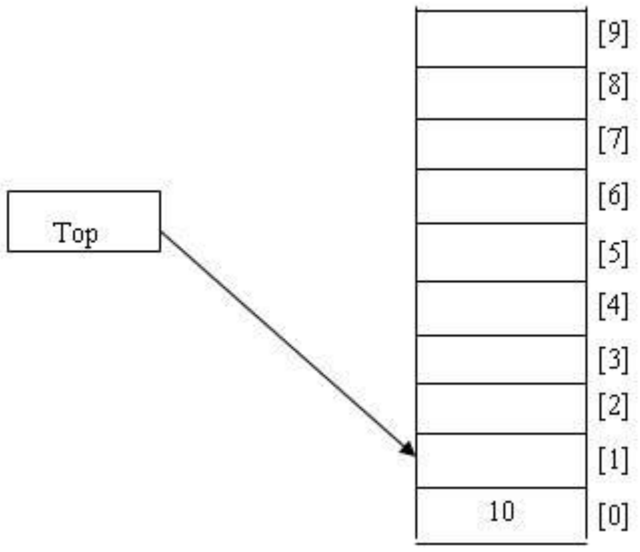
2)Pop Operation

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

Before :



After:



Algorithm For Push Operation

Step 1 − Checks if the stack is empty.

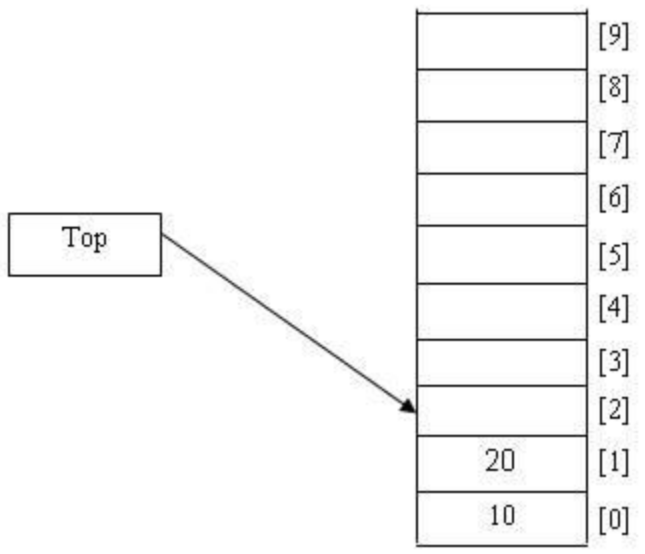
Step 2 − If the stack is empty, produces an error and exit.

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

Step 4 − Decreases the value of top by 1.

Step 5 − Returns success.

4)Peek or Top: get the top data element of the stack, without removing it



In this example it show top value=20

Algorithm of Peek operation

Step 1 − Checks if the stack is empty.

Step 2 − If the stack is empty, produces an error and exit.

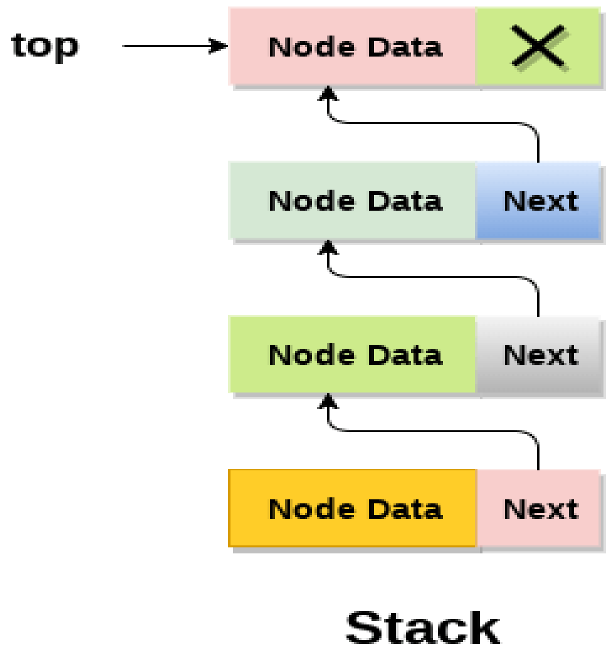
Step 3 − If the stack is not empty, accesses the data element at which top is pointing.

Step 4 − Returns success

Link List (Dynamic) Representation of Stack in Memory:

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

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



Stack underflow happens when we try to pop (remove) an item from the stack, when nothing is actually there to remove. This will raise an alarm of sorts in the computer because we told it to do something that cannot be done.

Stack overflow happens when we try to push one more item onto our stack than it can actually hold. You see, the stack usually can hold only so much stuff. Typically, we allocate (set aside) where the stack is going to be in memory and how big it can get. So, when we stick too much stuff there or try to remove nothing, we will generate a stack overflow condition or stack underflow condition, respectively.

1)Push:

```
Step1:[check for free space in memory]
  If AVAIL=NULL then
Write ”underflow” and Exit
```

```
Step2:[Allocate new memory from the Available memory]
  Set  NEW:=AVAIL, AVAIL:=LINK[AVAIL]
Step3:Set INFO[NEW]:=ITEM[Assign the item to new node]
Step4:Set LINK[NEW]:=TOP
Step6:Exit
```

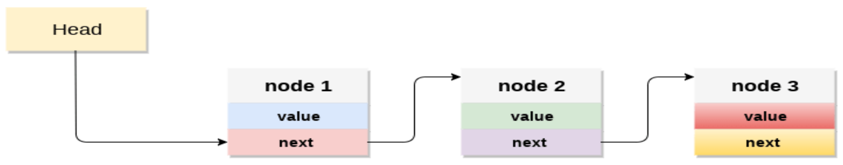
Adding a node to the stack (Push operation)

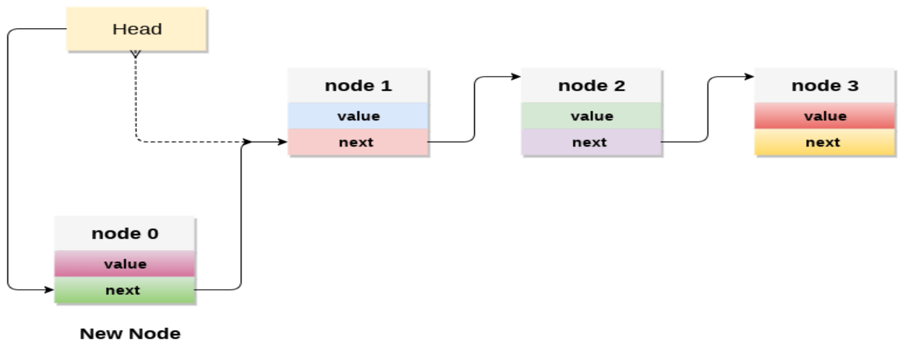
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

Create a node first and allocate memory to it.

If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.





1)Pop:

```
Step1:[check for Empty]
  If TOP=NULL then
Write "Underflow" and Exit
Step2:[Remove the top most element and copy to item]
   Set  ITEM:=INFO[TOP]
Step3:[Remember the old value of top and update top to next]
Set TEMP:=TOP,TOP:=LINK[TOP]
Step4:[Assign the deleted node to Avail]
Set LINK[TEMP]:=AVAIL , AVAIL:=TEMP
Step5:Exit
```

Deleting a node from the stack (POP operation)

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

Check for the underflow condition: The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.

Adjust the head pointer accordingly: In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

❖ Stack Applications:

Expression Evaluation

Stack is used to evaluate prefix, postfix and infix expressions.

Expression Conversion

An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

Syntax Parsing

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Backtracking

Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

Parenthesis Checking

Stack is used to check the proper opening and closing of parenthesis.

String Reversal

Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

Function Call

Stack is used to keep information about the active functions or subroutines.

❖ Multiple Stack

When a stack is created using single array, we cannot able to store large amount of data, thus this problem is rectified using more than one stack in the same array of sufficient array. This technique is called as Multiple Stack.

❖ Implement two stacks in an array (Multiple Stack)

Create a data structure two Stacks that represents two stacks. Implementation of two Stacks should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by two Stacks.
push1 (int x) –> pushes x to first stack
push2 (int x) –> pushes x to second stack

pop1 () –> pops an element from first stack and return the popped element
pop2() –> pops an element from second stack and return the popped element

Implementation of two Stack should be space efficient.

Method 1 (Divide the space in two halves)
A simple way to implement two stacks is to divide the array in two halves and assign the half half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Complexity Analysis:
- **Time Complexity:**
  - **Push operation :** O(1)
  - **Pop operation :** O(1)
- **Auxiliary Space:** O(N).
  Use of array to implement stack so. It is not the space-optimized method as explained above.

Method 2 (A space efficient implementation)
This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks. This check is highlighted in the below code.

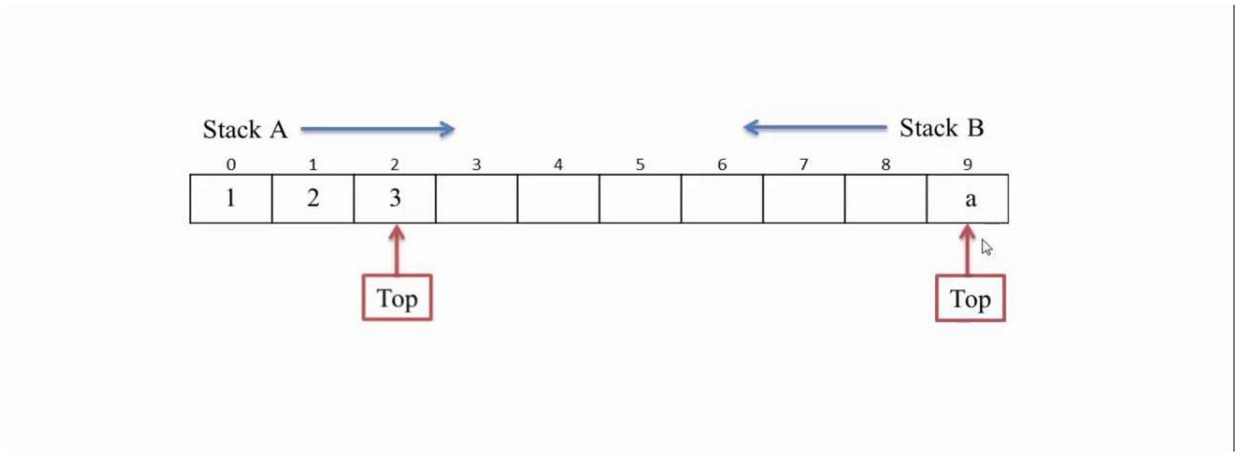


Fig .- Multiple Stack

Fig. Single Stack

❖ Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. **Examples** of such problems are Towers of Hanoi ,Conversion from one notation to another notation

Example of

- Factorial of a positive number
- Fibonacci series
- Towers of Hanoi

1)Factorial of a positive number:

This function returns n! (read n factorial) where n is an integer.

n!=n* n-1* n-2*....2*1 by definition 0! is 1

for example

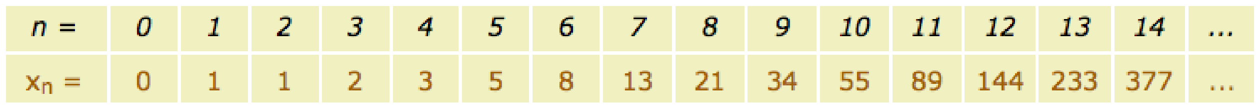
if n==5, then n! would be 5! = 5*4*3*2*1=120

Algorithm

```
STEP 1:IF n=1 THEN
  Set f:=1
  ELSE
  Set f=n*factorial(n-1)
    [END IF]
Step2: Return F
Step 3:End
```

2)Fibonacci series

Fibonacci series is defined as a sequence of numbers in which the first two numbers are 1 and 1, or 0 and 1, depending on the selected beginning point of the sequence, and each subsequent number is the sum of the previous two.



n =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
x_n =	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	...

First Term = 0
Second term = 1
Third Term = First + Second = 0+1 =1
Fourth term = Second + Third =1+1 = 2
Fifth Term = Third + Fourth = 2+1 = 3
Sixth Term= Fourth + Fifth = 3+2 = 5
Seventh Term = Fifth + Sixth = 3+5 = 8
Eighth Term = Sixth + Seventh = 5+8 = 13 … and so on to infinity!

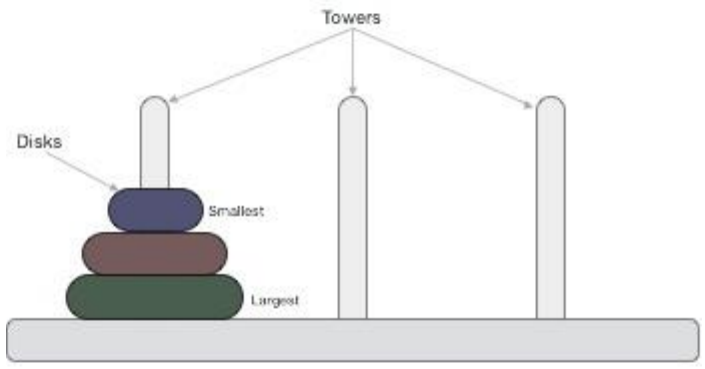
Algorithm

```
STEP 1:IF n=0 THEN
  Set f:=0
ELSE IF n=1 then
 Set f:=1
  ELSE
  Set f=fib(n-1)+fib(n-2)
    [END IF]
Step2 :Return F
Step 3:End
```

3)Towers of Hanoi

Tower of Hanoi is a mathematical puzzle with three rods and 'n' numbers of discs.

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

Only one disk can be moved among the towers at any given time.

Only the "top" disk can be removed.

No large disk can sit over a small disk.

Algorithm :

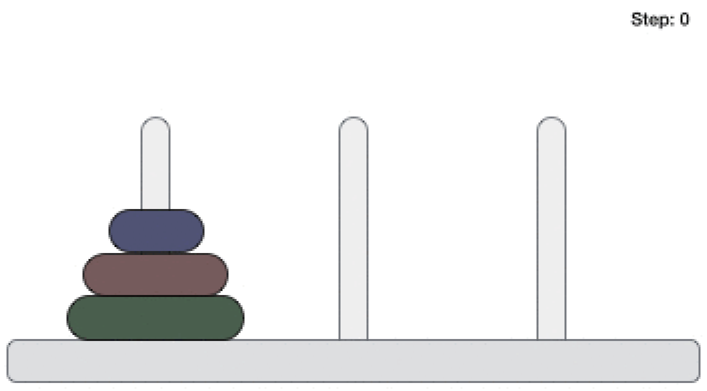
The steps to follow are

Step 1 − Move n-1 disks from source to aux

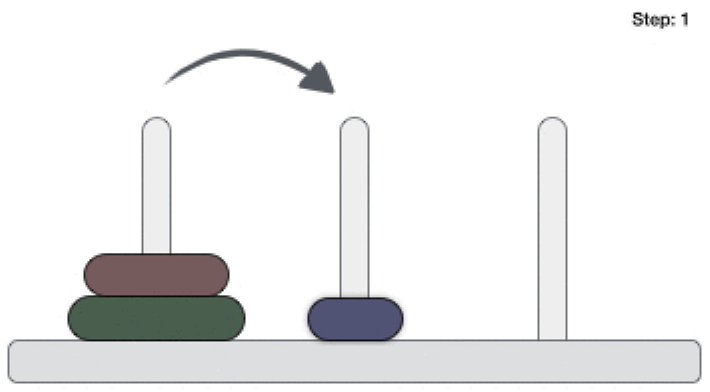
Step 2 − Move nth disk from source to dest

Step 3 − Move n-1 disks from aux to dest

Step 1:

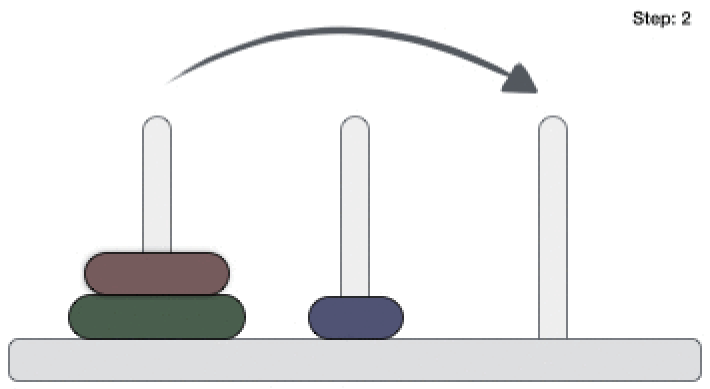


Step 2:

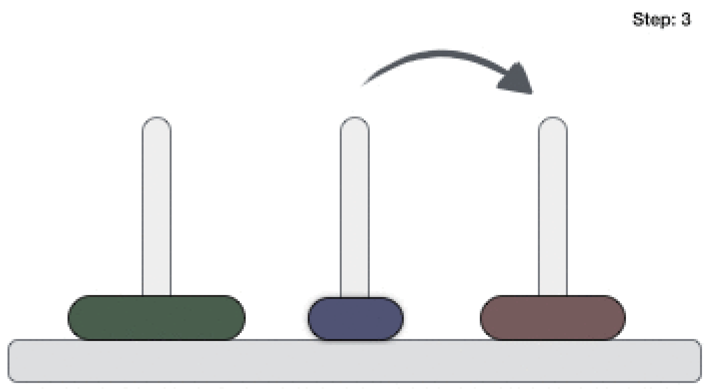


Step 3:

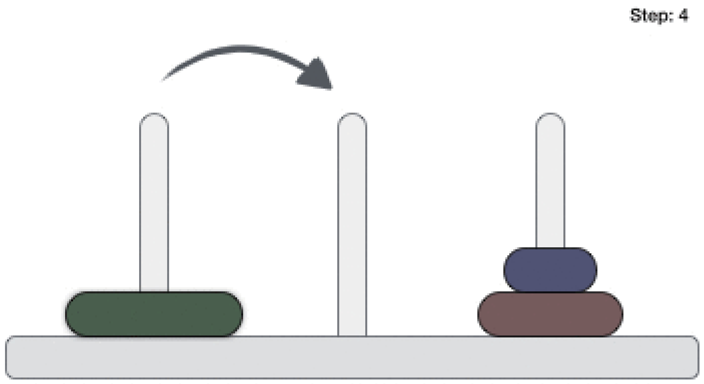
[illegible]



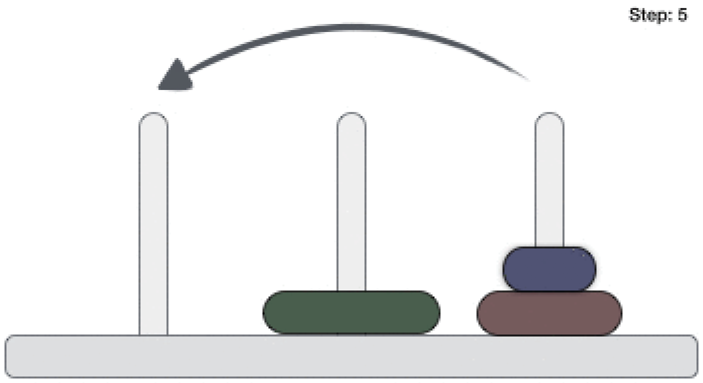
Step 4:



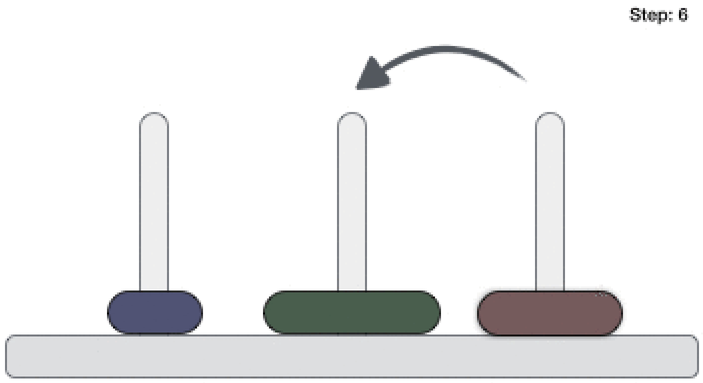
Step 5:



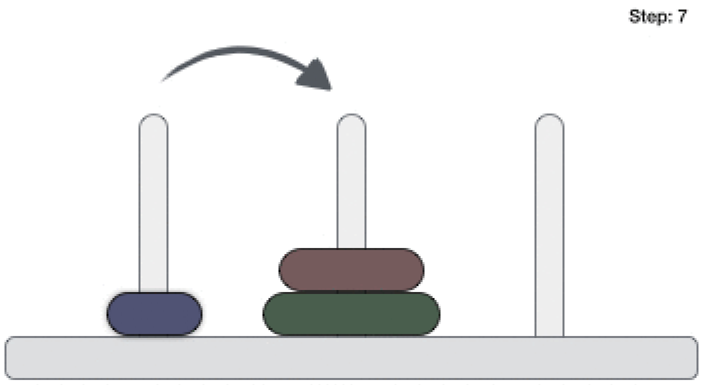
Step 6:



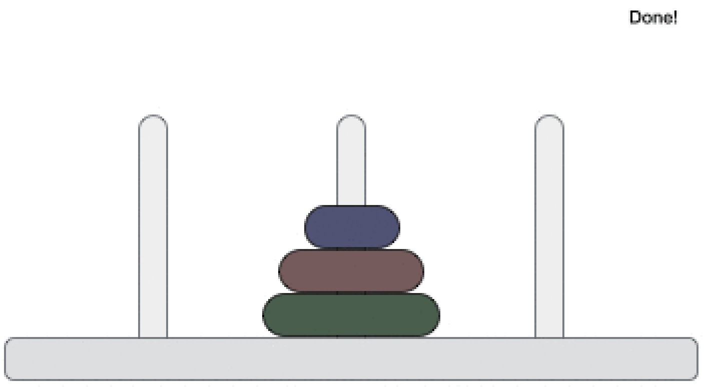
Step 7:



Step8:



Step9:



❖ Conversion from one notation to another notation

1)Convert Infix To Prefix Notation

While we use infix expressions in our day to day lives. Computers have trouble understanding this format because they need to keep in mind rules of operator precedence and also brackets. Prefix and Postfix expressions are easier for a computer to understand and evaluate.

Examples:

Input : A * B + C / D

Output : + * A B/ C D

Input : (A - B/C) * (A/K-L)

Output : *-A/BC-/AKL

To convert an infix to postfix expression refer to this article Stack | Set 2 (Infix to Postfix). We use the same to convert Infix to Prefix.

Step 1: Reverse the infix expression i.e A+B*C will become C*B+A. Note while reversing each '(' will become ')' and each ')' becomes '('.

Step 2: Obtain the postfix expression of the modified expression i.e CB*A+.

Step 3: Reverse the postfix expression. Hence in our example prefix is +A*BC.

2)Infix to Prefix conversion using two stacks

Infix : An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).
Example : (A+B) * (C-D)

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).
Example : *+AB-CD (Infix : (A+B) * (C-D))

Given an Infix expression, convert it into a Prefix expression using two stacks.

Examples:

Input : A * B + C / D

Output : + * A B/ C D

Input : (A - B/C) * (A/K-L)

Output : *-A/BC-/AKL

The idea is to use one stack for storing operators and other to store operands. The stepwise algo is:

Traverse the infix expression and check if given character is an operator or an operand.

If it is an operand, then push it into operand stack.

If it is an operator, then check if priority of current operator is greater than or less than or equal to the operator at top of the stack. If priority is greater, then push operator into operator stack. Otherwise pop two operands from operand stack, pop operator from operator stack and push string operator + operand1 + operand 2 into operand stack. Keep popping from both stacks and pushing result into operand stack until priority of current operator is less than or equal to operator at top of the operator stack.

If current character is '(', then push it into operator stack.

If current character is ')', then check if top of operator stack is opening bracket or not. If not pop two operands from operand stack, pop operator from operator stack and push string operator + operand1 + operand 2 into operand stack. Keep popping from both stacks and pushing result into operand stack until top of operator stack is an opening bracket.

The final prefix expression is present at top of operand stack.

❖ Queues:

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

LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

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

❖ Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

enqueue() − add (store) an item to the queue.

dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

peek() − Gets the element at the front of the queue without removing it.

isfull() − Checks if the queue is full.

isempty() − Checks if the queue is empty.

Comparison between Stack and Queue

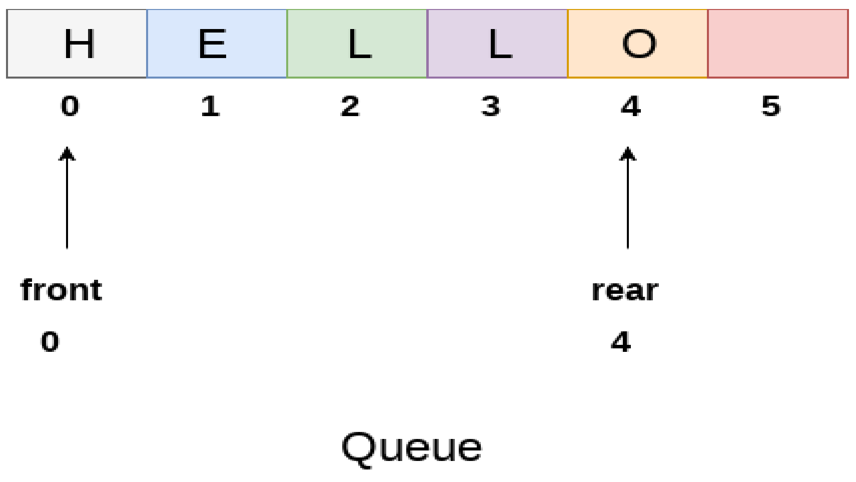
STACKS	QUEUES
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.

STACKS	QUEUES
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.

❖ Memory Representation of Queue in Memory

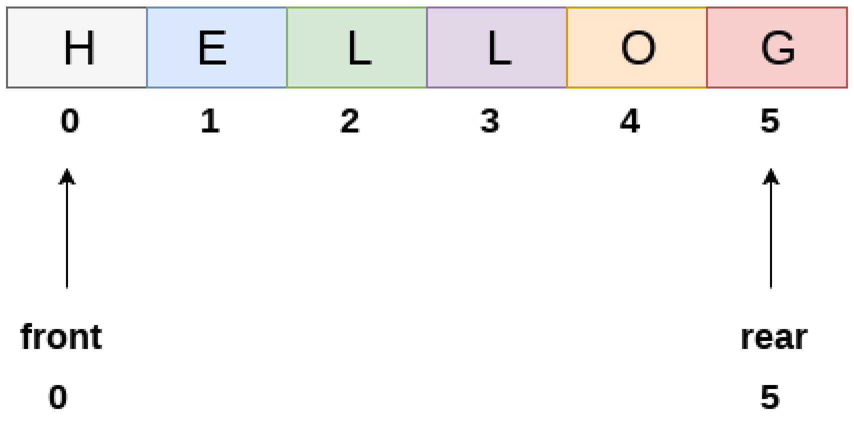
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



1)Insertion in Queue using Array

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

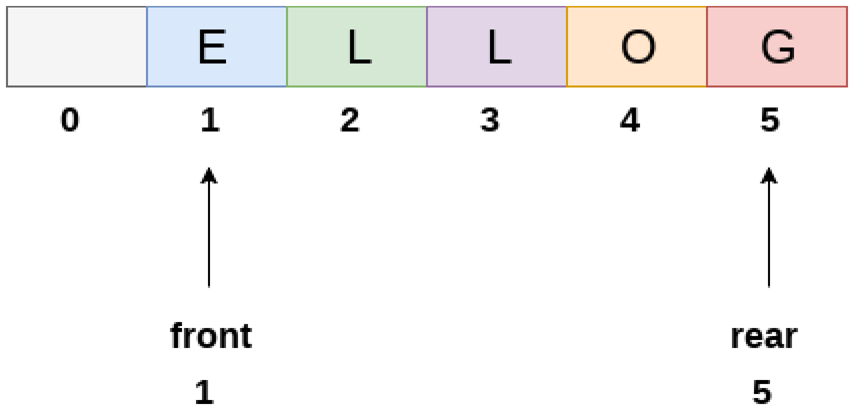
Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
Step 1: IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]

Step 3: Set QUEUE[REAR] = NUM

Step 4: EXIT
```

2)Deletion in Queue using Array

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

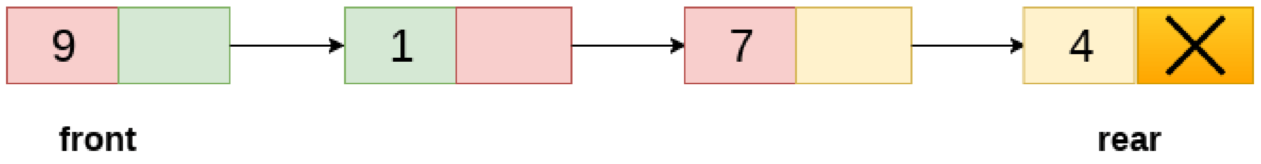
Algorithm

```
Step 1: IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]

Step 2: EXIT
```

❖ **Memory Representation of Queue in Memory**

Linked List implementation of Queue

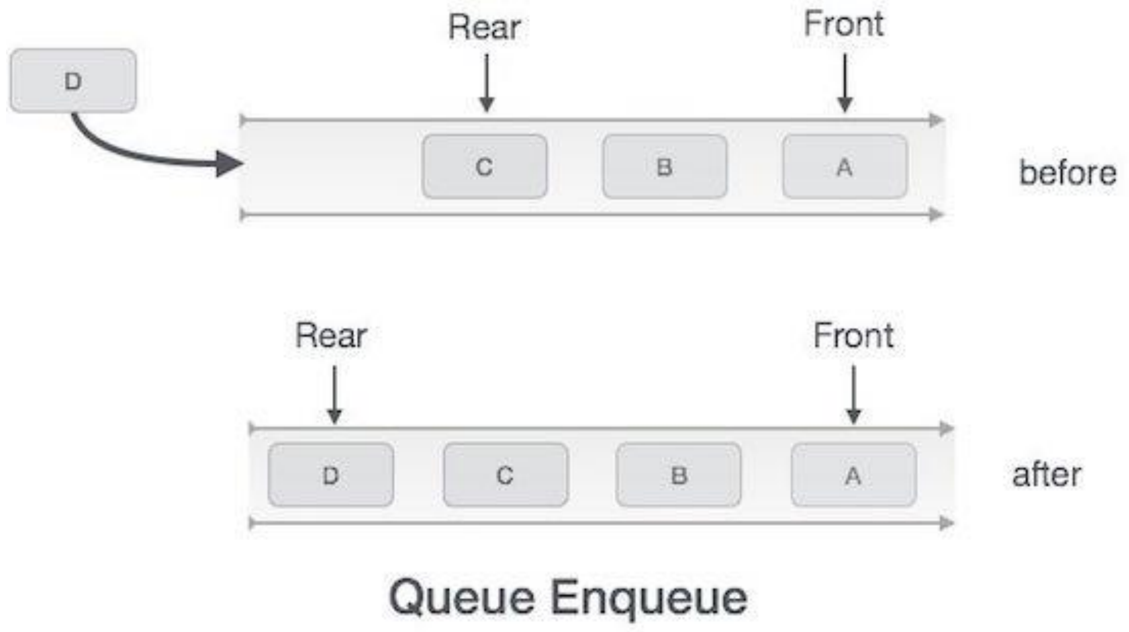
Due to the drawbacks , the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.



Linked Queue

1)Insertion in Queue using LinkList

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.



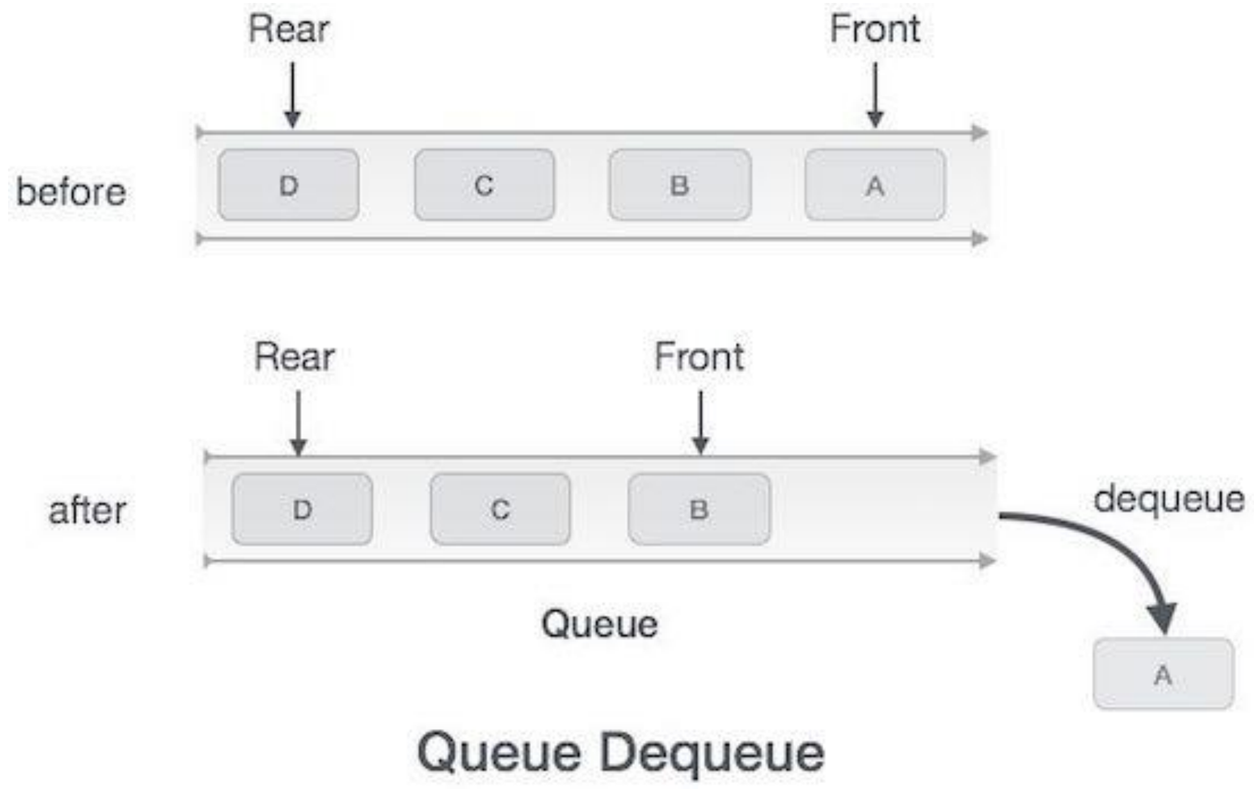
Algorithm

```
Step 1: Allocate the space for the new node PTR

Step 2: SET PTR -> DATA = VAL

Step 3: IF FRONT = NULL
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]

Step 4: END
```

2)Deletion in Queue using LinkList

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr.



The algorithm and C function is given as follows.

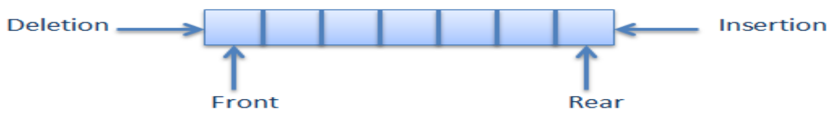
Algorithm

```
Step 1: IF FRONT = NULL
Write " Underflow "
Go to Step 5
[END OF IF]
```

```
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

❖ Types of Queues in Data Structure

Simple Queue



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

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

Simple queue insertion using array(static queue)

Here q is an array with n elements, FRONT and REAR points to the front and rear of the queue. ITEM is the value to be inserted. This algorithm inserts an ITEM in the simple queue.

```
Step1: If ( REAR=N) then (check overflow)
        write "OVERFLOW" and EXIT
(end if)
Step 2:  IF ( FRONT=NULL and rear = NULL) then
```

```
            (check if QUEUE is empty)
Set FRONT:= 1, REAR:=1
Else
    Set REAR = REAR+1 (Increment REAR by 1)
[End if]
Step 3: Q[REAR] = ITEM
Step 4: Exit
```

Simple queue deletion using array(Static queue)

Here q is an array with N elements, FRONT and RAER points to the front and rear of the QUEUE. It deletes the element from front of the queue and assign to variable ITEM

```
Step 1: If(FRONT = NULL) then [check for underflow)
      write "Underflow " and exit
[exit]
Step2: Set ITEM := Q[ FRONT]
Step3 : If (FRONT= REAR ) then
                (check if only one element is left)
 Set FRONT := NULL
Set REAR := NULL
Else
Set FRONT:= FRONT+1 [increment by 1]
[End if]
Step 4: exit
```

Simple queue insert using Linked List(Dynamic queue)

Here Q is a queue using linked list and we have to insert an item, front and rear are two pointers used to manage queue insertion and deletion operation. Avail is the free pool of memory from where new node for insertion allocated. This algorithm inserts the item in queue Q.

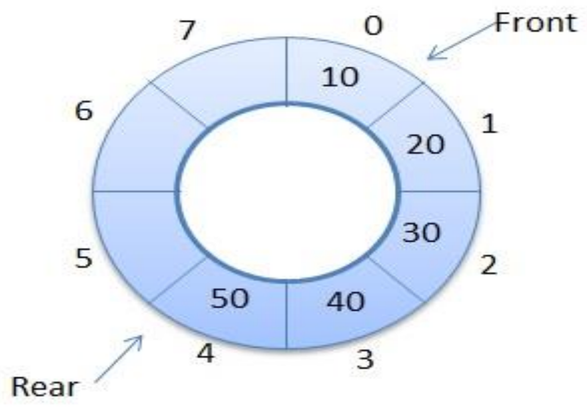
```
Step1: If Avail = null then
  write “overflow “ and exit
   [End if]
Step2: Set TEMP:= AVAIL, AVAIL:= LINK[AVAIL],
         Set INFO[TEMP]:= ITEM, LINK[TEMP]: =NULL
Step3: IF FRONT = NULL Then
      Set FRONT := TEMP, REAR := TEMP
Else
    Set LINK [REAR]:= TEMP,
    Set REAR := TEMP
[End IF]
Step 4: Exit
```

Simple Queue Deletion using Linked List(Dynamic Queue)

Here Q is a queue using linked list and we have to insert an item, front and rear are two pointers used to manage queue insertion and deletion operation. Avail is the free pool of memory from where deleted node is attached . This algorithm deletes the front node and information is stored in item.

```
Step 1: IF FRONT =NULL hen
     write “ underflow” and exit
Step 2: Set ITEM = INFO [FRONT]
Step 3: If FRONT = REAR  then
            set TEMP:= FRONT
Set Front:= NULL and REAR:= Null
    Else
       Set TEMP:= FRONT
       Set FRONT:= LINK[FRONT]
[End if]
Step4: Set LINK[TEMP]:= AVAIL
            AVAIL := TEMP
Step 5 : EXIT
```

Circular Queue



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

Algorithm :- Circular queue insert using array (static queue)

Here Q is a circular queue using linear array with size of N and we have to insert an item, front and REAR are two index uses to manage the pointer for insertion and deletion operation. This algorithm inserts the item in circular queue Q at appropriate position.

```
Step : IF (FRONT =1 and REAR =N ) or (FRONT = REAR+1) then
                Write "OVERFLOW" and EXIT       [overflow check]
Step :  IF FRONT = NULL Then
                Set FRONT : =1 and  REAR : =1
 ELSE IF REAR =N Then
        Set REAR : = 1
Else
        Set REAR : =1
[END IF]
Step : 3  Set  Q [REAR] : = ITEM
Step :4 EXIT
```

Algorithm : Circular queue deletion using array (Static Queue)

Here Q is a circular Queue using linear array with size N and we have to delete an item, Front, Rear are twp index uses to manage the pointer for insertion and deletion operation. This algorithm deletes element from front and assign it in the item.

```
Step : IF FRONT = NULL  Then [Empty queue check]
write “underflow “ and Exit
[End IF]
Step 2 : Set Item : = Q [FRONT]
Set  FRONT := REAR : = NULL
Else IF FRONT = N Then
        Set FRONT : = 1
Else
Set Front : = Front + 1
[END IF]
Step 4 : EXIT
```

❖ The benefit of a circular queue:

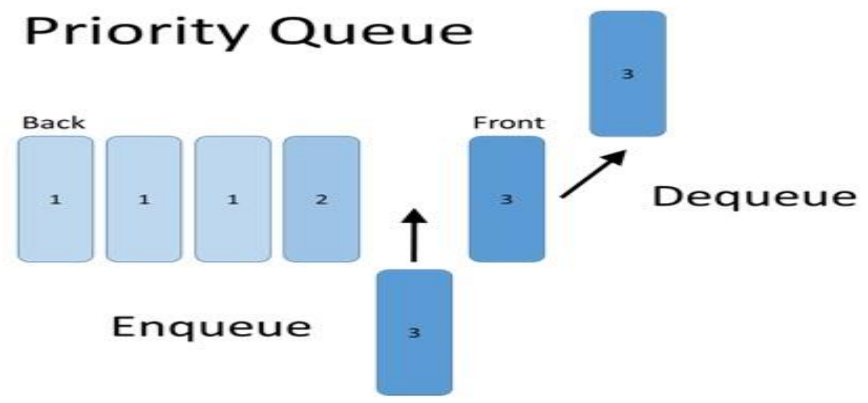
1. In circular queue, memory is utilized. If we delete any element that position is used later, because it is circular.

2. Circular queue consumes less memory than linear queue because in queue while doing insertion after deletion operation it allocate an extra space the first remaining vacant but in circular queue the first is used as it comes immediate after the last.

3. In CQ the memory of the deleted process can be used by some other new process.

4. A standard queue suffers from a rebuffering problem during deque operations. By making the queue circular and linking the head to the tail, this alleviates the problem and allows insertion and deletion in constant time.

Priority Queue



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

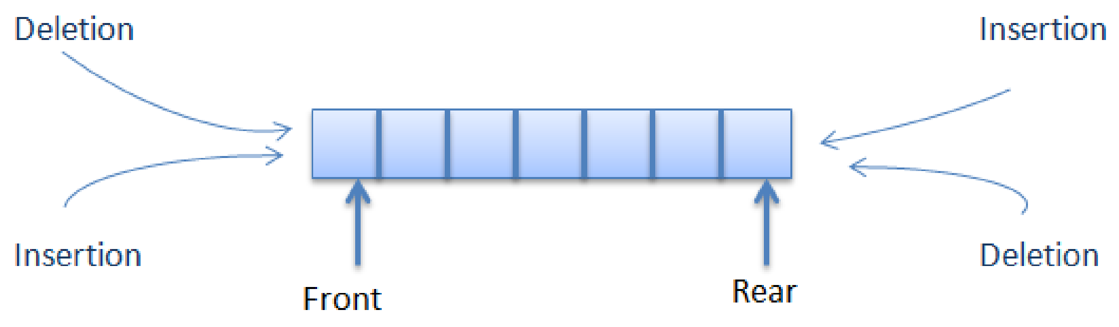
While the deletion is performed in accordance to priority number (the data item with highest priority is removed first), insertion is performed only in the order.

Comparison Chart

BASIS FOR COMPARISON	LINEAR QUEUE	CIRCULAR QUEUE
Basic	Organizes the data elements and instructions in a sequential order one after the other.	Arranges the data in the circular pattern where the last element is connected to the first element.

BASIS FOR COMPARISON	LINEAR QUEUE	CIRCULAR QUEUE
Order of task execution	Tasks are executed in order they were placed before (FIFO).	Order of executing a task may change.
Insertion and deletion	The new element is added from the rear end and removed from the front.	Insertion and deletion can be done at any position.
Performance	Inefficient	Works better than the linear queue.

❖ Doubly Ended Queue (Dequeue)



Double ended queue is a linear list of element , in which all insertion and deletion are made at the either end of the list. A dequeue is pronounced as “deck” or “ de queue”.

In simple words we can say that dequeue is special types of queue in which insertion and deletion operation can be performed on either end i.e. We can insert element in the queue from front as well as rear but not in middle of the queue and deletion operation can also be performed on any end i.e we can delete element from front as we as from rear position of the queue but not in middle.

There are two types of deque

- Input restricted deque
- Output restricted deque

Input Restricted Deque

In this type of queue insertion can be performed on only one end but deletion can be performed on both end of the queue.

Output Restricted Deque

In this type of deque array with two variable on both end but deletion can be performed only on one end.

A deque is commonly implemented as a circular array with two variable LEFT and RIGHT taking care of the active ends of the queue.

to understand the wirking of deque, consider the following example in which we assume that insertion and deletion is applicable on both end.

❖ Implementation of Double ended Queue

Here we will implement a double ended queue using a circular array. It will have the following methods:

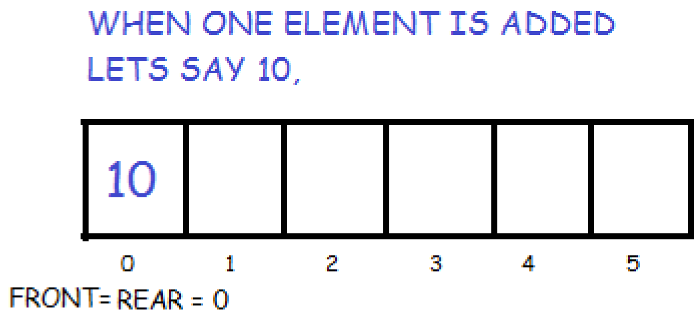
- push_back : inserts element at back

- push_front : inserts element at front
- pop_back : removes last element
- pop_front : removes first element
- get_back : returns last element
- get_front : returns first element
- empty : returns true if queue is empty
- full : returns true if queue is full

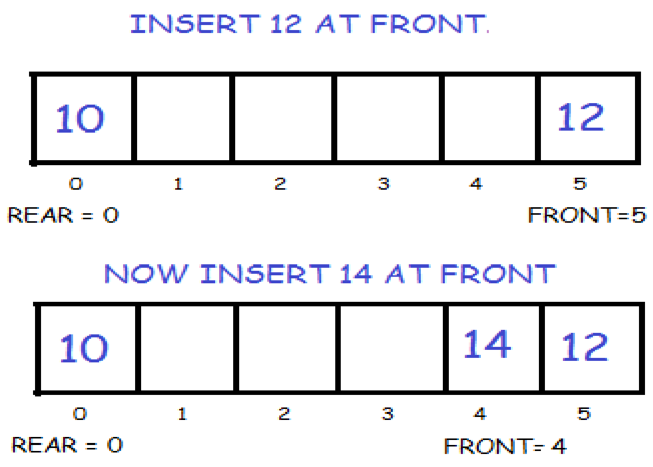
Insert Elements at Front

First we check if the queue is full. If its not full we insert an element at front end by following the given conditions :

If the queue is empty then intialize front and rear to 0. Both will point to the first element.



Else we decrement front and insert the element. Since we are using circular array, we have to keep in mind that if front is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.

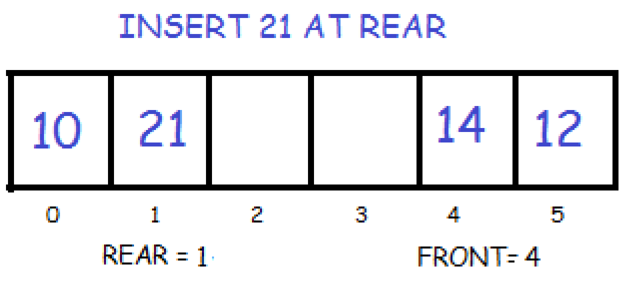


Insert Elements at back

Again we check if the queue is full. If its not full we insert an element at back by following the given conditions:

If the queue is empty then initialize front and rear to 0. Both will point to the first element.

Else we increment rear and insert the element. Since we are using circular array, we have to keep in mind that if rear is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.

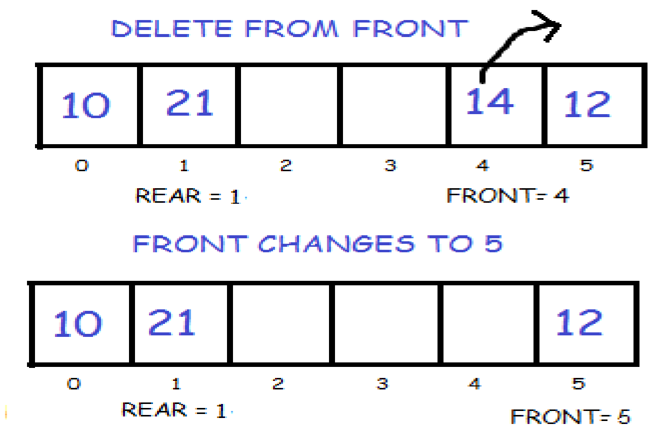


Delete First Element

In order to do this, we first check if the queue is empty. If its not then delete the front element by following the given conditions :

If only one element is present we once again make front and rear equal to -1.

Else we increment front. But we have to keep in mind that if front is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.

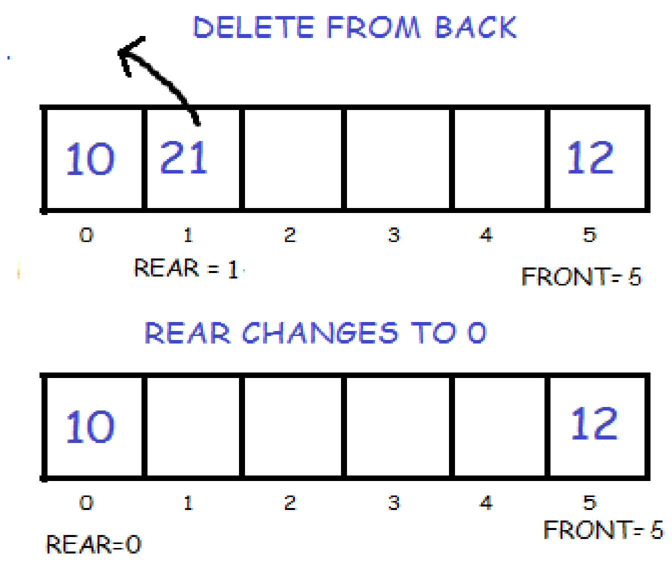


Delete Last Element

Inorder to do this, we again first check if the queue is empty. If its not then we delete the last element by following the given conditions :

If only one element is present we make front and rear equal to -1.

Else we decrement rear. But we have to keep in mind that if rear is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.



❖ Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

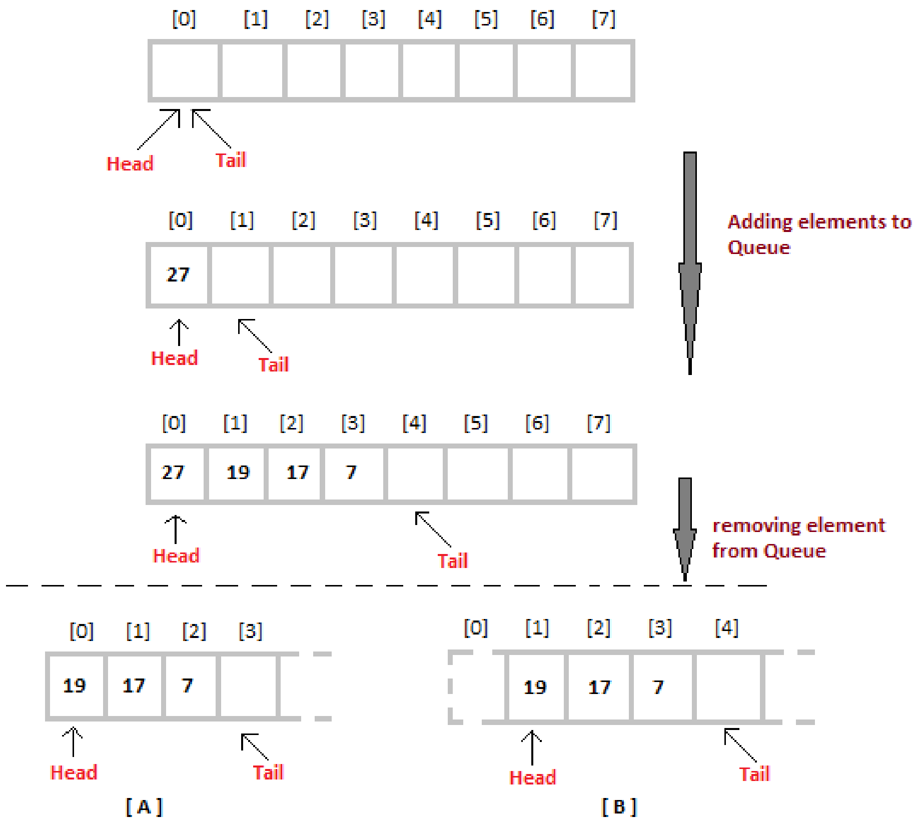
Initially the head(FRONT) and the tail(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.

When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at head position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from head position and then move head to the next position.

In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the size on Queue is reduced by one space each time.



Algorithm for ENQUEUE operation

Check if the queue is full or not.

If the queue is full, then print overflow error and exit the program.

If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

Check if the queue is empty or not.

If the queue is empty, then print underflow error and exit the program.

If the queue is not empty, then print the element at the head and increment the head

❖ Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.

Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e. First come first served.

UNIT III

❖ Linked Lists

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Linked List** − A Linked List contains the connection link to the first link called First.

Types of Linked List

Following are the various types of linked list.

- **Singly Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

Following are the basic operations supported by a list.

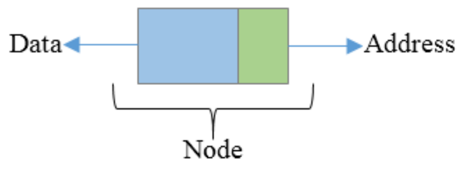
- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

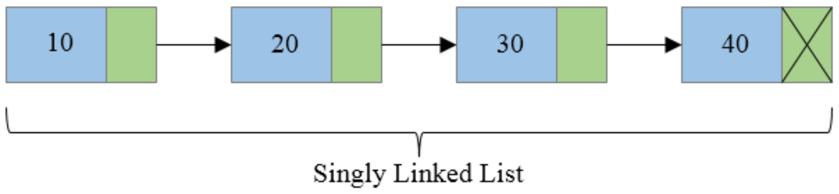
1) Singly Linked List − Singly Linked Lists are a type of data structure. A linked list provides an alternative to an array-based structure.

A linked list, in its simplest form, in a collection of **nodes** that collectively form linear sequence.

In a **singly linked** list, each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list. It does not store any pointer or reference to the previous node.

To store a single linked list, only the reference or pointer to the first node in that list must be stored. The last node in a single linked list points to nothing.





Basic Operations

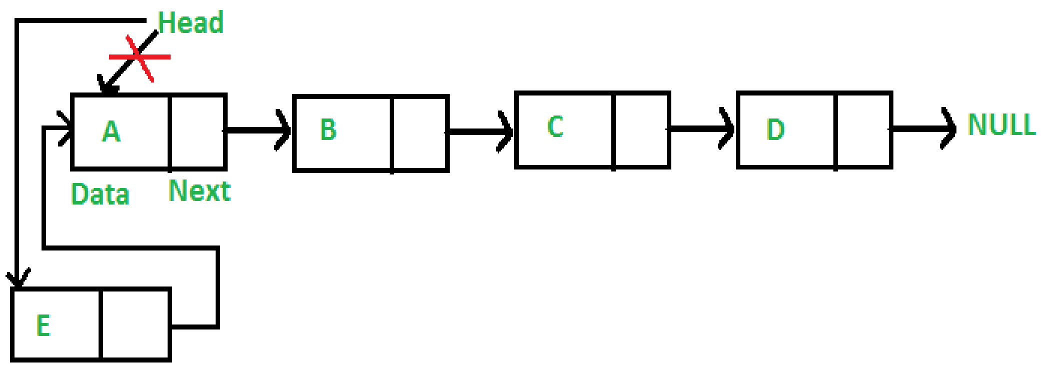
Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.

  A node can be added in three ways

  1) At the front of the linked list:

  The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. **For example** if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25.

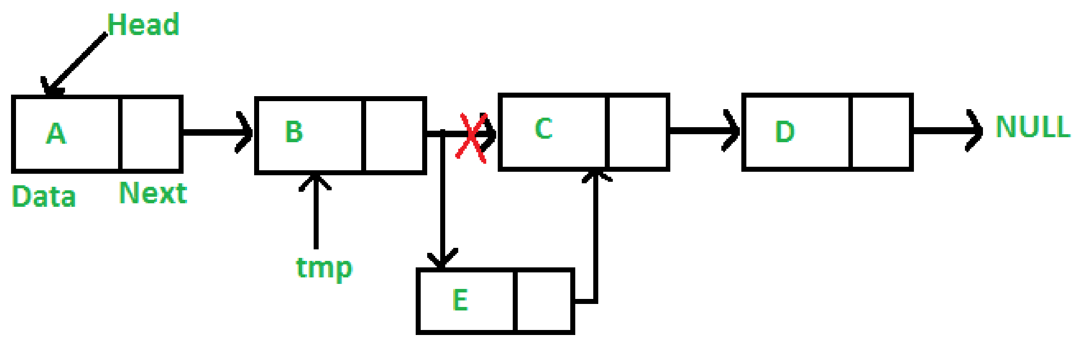


Algorithm

```
• Step 1: IF PTR = NULL
  Write OVERFLOW
     Go to Step 7
    [END OF IF]
• Step 2: SET NEW_NODE = PTR
• Step 3: SET PTR = PTR → NEXT
• Step 4: SET NEW_NODE → DATA = VAL
• Step 5: SET NEW_NODE → NEXT = HEAD
• Step 6: SET HEAD = NEW_NODE
```

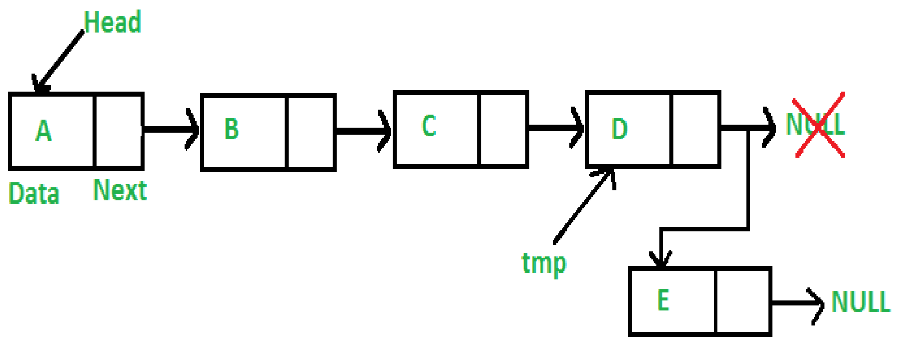
- Step 7: EXIT

2) After a given node: We are given pointer to a node, and the new node is inserted after the given node**.**



- **STEP 1:** IF PTR = NULL
  WRITE OVERFLOW
  GOTO STEP 12
  END OF IF
- **STEP 2:** SET NEW_NODE = PTR
- **STEP 3:** NEW_NODE → DATA = VAL
- **STEP 4:** SET TEMP = HEAD
- **STEP 5:** SET I = 0
- **STEP 6:** REPEAT STEP 5 AND 6 UNTIL I<loc< li=""> </loc<>
- **STEP 7:** TEMP = TEMP → NEXT
- **STEP 8:** IF TEMP = NULL
  WRITE "DESIRED NODE NOT PRESENT"
  GOTO STEP 12
  END OF IF
  END OF LOOP
- **STEP 9:** PTR → NEXT = TEMP → NEXT
- **STEP 10:** TEMP → NEXT = PTR
- **STEP 11:** SET PTR = NEW_NODE
- **STEP 12:** EXIT

3) At the end of the linked list: The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.
Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



```
• Step 1: IF PTR = NULL
  Write OVERFLOW
     Go to Step 1
    [END OF IF]
• Step 2: SET NEW_NODE = PTR
• Step 3: SET PTR = PTR - > NEXT
• Step 4: SET NEW_NODE - > DATA = VAL
• Step 5: SET NEW_NODE - > NEXT = NULL
• Step 6: SET PTR = HEAD
• Step 7: Repeat Step 8 while PTR - > NEXT != NULL
• Step 8: SET PTR = PTR - > NEXT
  [END OF LOOP]
• Step 9: SET PTR - > NEXT = NEW_NODE
• Step 10: EXIT
```

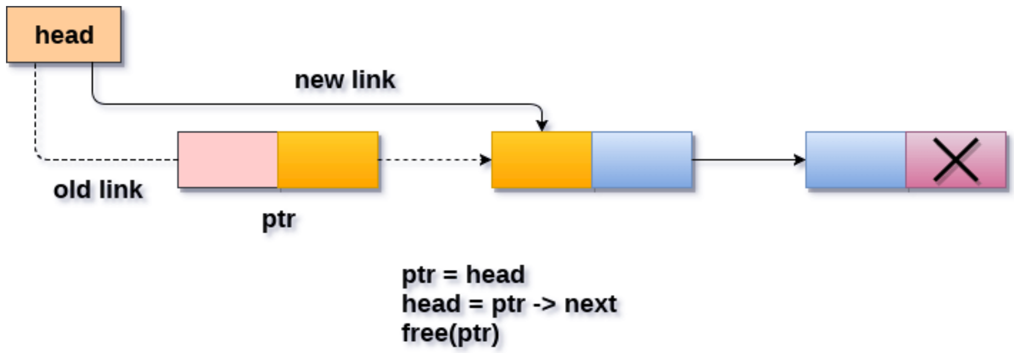
Deletion

1) Deletes an element at the beginning of the list.

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head.

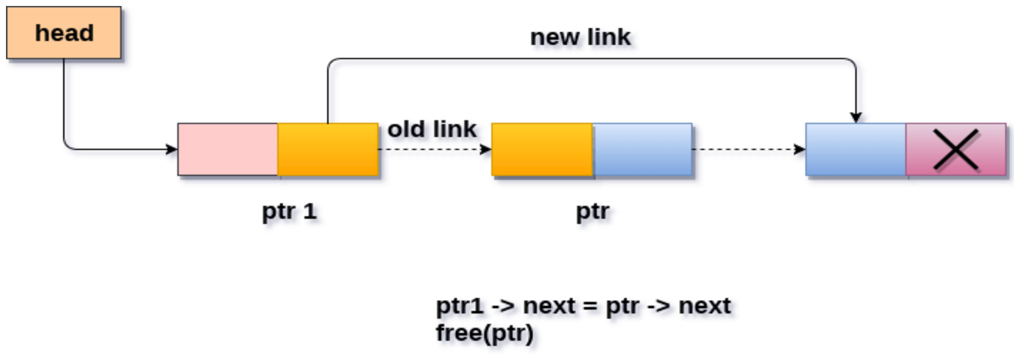
```
• Step 1: IF HEAD = NULL
Write UNDERFLOW
    Go to Step 5
   [END OF IF]
• Step 2: SET PTR = HEAD
```

- **Step 3:** SET HEAD = HEAD -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** EXIT



Deleting a node from the beginning

2) Deletes an element at the given location of the list.
In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node.



Deletion a node from specified position

Algorithm

```
• STEP 1: IF HEAD = NULL
WRITE UNDERFLOW
    GOTO STEP 10
   END OF IF
```

- **STEP 2:** SET TEMP = HEAD
- **STEP 3:** SET I = 0
- **STEP 4:** REPEAT STEP 5 TO 8 UNTIL I<loc< li=""> </loc<>
- **STEP 5:** TEMP1 = TEMP
- **STEP 6:** TEMP = TEMP → NEXT
- **STEP 7:** IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT"
    GOTO STEP 12
    END OF IF

- **STEP 8:** I = I+1

END OF LOOP

- **STEP 9:** TEMP1 → NEXT = TEMP → NEXT
- **STEP 10:** FREE TEMP
- **STEP 11:** EXIT

3) Deletes an element at the end of the list.

There are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

Algorithm

- **Step 1:** IF HEAD = NULL

Write UNDERFLOW
   Go to Step 8
  [END OF IF]

- **Step 2:** SET PTR = HEAD
- **Step 3:** Repeat Steps 4 and 5 while PTR -> NEXT!= NULL
- **Step 4:** SET PREPTR = PTR
- **Step 5:** SET PTR = PTR -> NEXT

[END OF LOOP]

- **Step 6:** SET PREPTR -> NEXT = NULL
- **Step 7:** FREE PTR
- **Step 8:** EXIT

Display − Displays the complete list. Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that.

Algorithm

```
• STEP 1: SET PTR = HEAD
• STEP 2: IF PTR = NULL
     WRITE "EMPTY LIST"
    GOTO STEP 7
    END OF IF
• STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL
• STEP 5: PRINT PTR→ DATA
• STEP 6: PTR = PTR → NEXT
  [END OF LOOP]
• STEP 7: EXIT
```

Search − **Searches an element using the given key.**

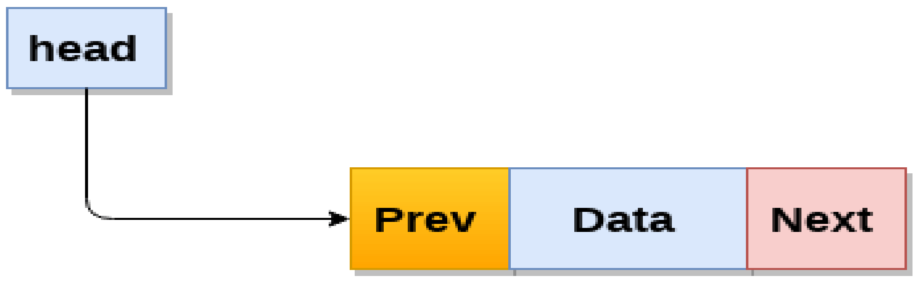
Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm

```
    • Step 1: SET PTR = HEAD
    • Step 2: Set I = 0
    • STEP 3: IF PTR = NULL
 WRITE "EMPTY LIST"
 GOTO STEP 8
 END OF IF
    • STEP 4: REPEAT STEP 5 TO 7 UNTIL PTR != NULL
    • STEP 5: if ptr → data = item
 Write i+1
End of IF
    • STEP 6: I = I + 1
    • STEP 7: PTR = PTR → NEXT
[END OF LOOP]
    • STEP 8: EXIT
```

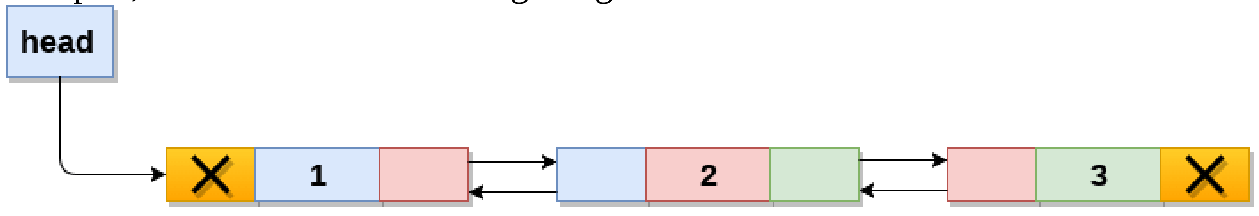
❖ Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



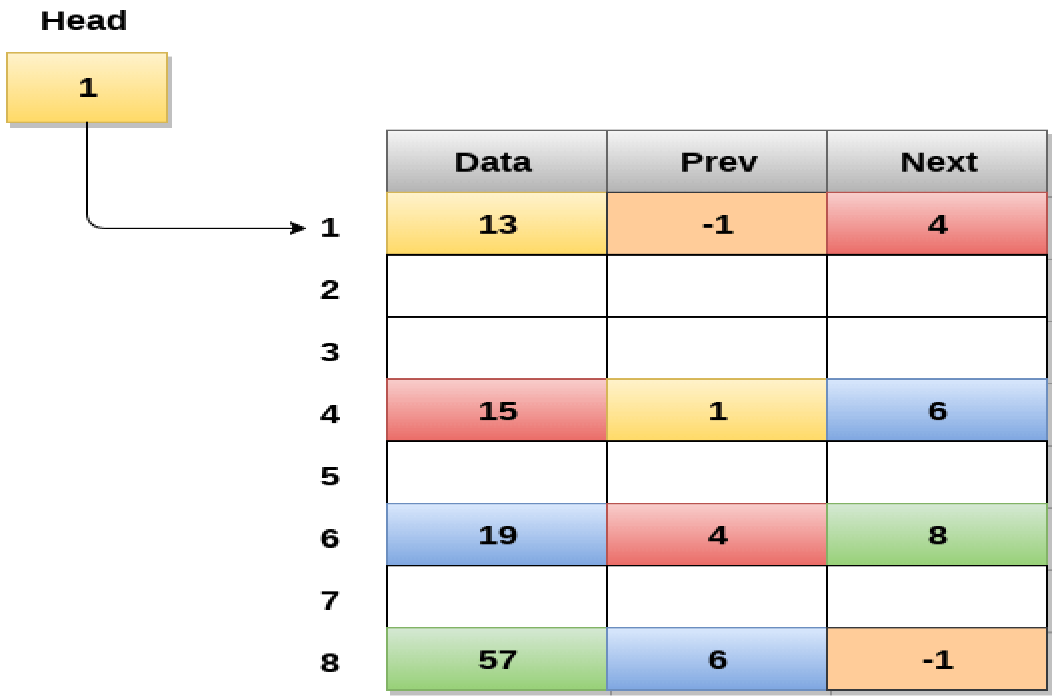
Doubly Linked List

❖ Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

❖ **Basic Operations**

Following are the basic operations supported by a list.

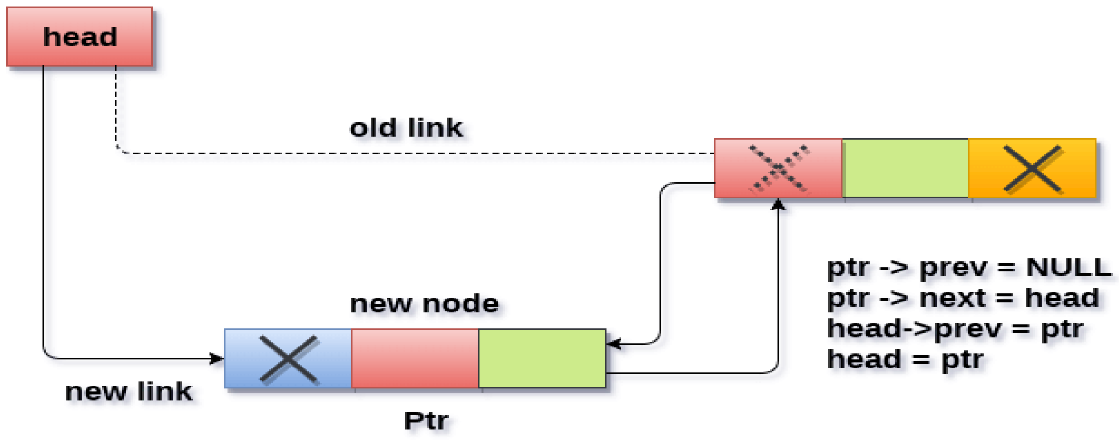
- **Insertion** –

  1)Adds an element at the beginning of the list.

  Algorithm :

- **Step 1:** IF ptr = NULL
  Write OVERFLOW
  Go to Step 9
  [END OF IF]
- **Step 2:** SET NEW_NODE = ptr
- **Step 3:** SET ptr = ptr -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> PREV = NULL
- **Step 6:** SET NEW_NODE -> NEXT = START

- **Step 7:** SET head -> PREV = NEW_NODE
- **Step 8:** SET head = NEW_NODE
- **Step 9:** EXIT



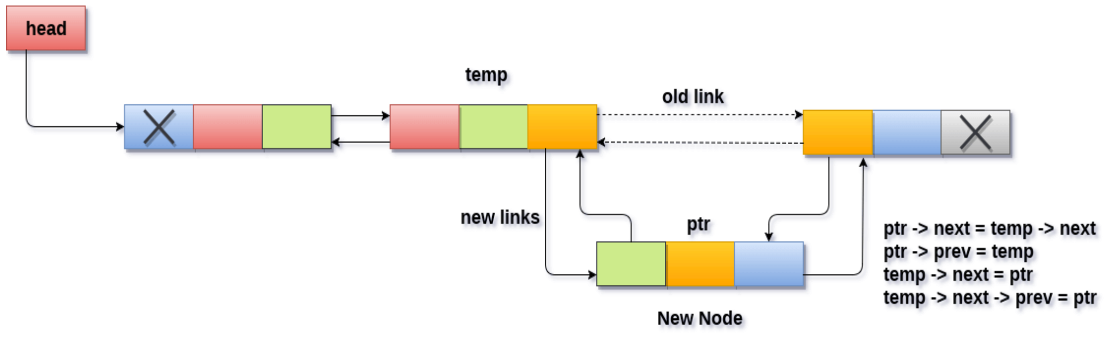
Insertion into doubly linked list at beginning

2)Adds an element at the given location of the list.

Algorithm

```
• Step 1: IF PTR = NULL
    Write OVERFLOW
    Go to Step 15
  [END OF IF]
• Step 2: SET NEW_NODE = PTR
• Step 3: SET PTR = PTR -> NEXT
• Step 4: SET NEW_NODE -> DATA = VAL
• Step 5: SET TEMP = START
• Step 6: SET I = 0
• Step 7: REPEAT 8 to 10 until I<="" li="">
• Step 8: SET TEMP = TEMP -> NEXT
• STEP 9: IF TEMP = NULL
• STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
    GOTO STEP 15
    [END OF IF]
  [END OF LOOP]
• Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT
• Step 12: SET NEW_NODE -> PREV = TEMP
• Step 13 : SET TEMP -> NEXT = NEW_NODE
```

- **Step 14:** SET TEMP -> NEXT -> PREV = NEW_NODE
- **Step 15:** EXIT
-

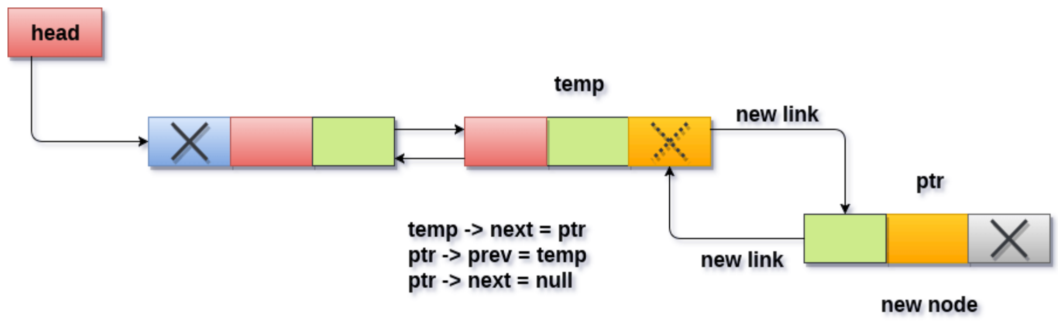


Insertion into doubly linked list after specified node

3)Adds an element at the end location of the list.

Algorithm

```
• Step 1: IF PTR = NULL
   Write OVERFLOW
    Go to Step 11
   [END OF IF]
• Step 2: SET NEW_NODE = PTR
• Step 3: SET PTR = PTR -> NEXT
• Step 4: SET NEW_NODE -> DATA = VAL
• Step 5: SET NEW_NODE -> NEXT = NULL
• Step 6: SET TEMP = START
• Step 7: Repeat Step 8 while TEMP -> NEXT != NULL
• Step 8: SET TEMP = TEMP -> NEXT
  [END OF LOOP]
• Step 9: SET TEMP -> NEXT = NEW_NODE
• Step 10C: SET NEW_NODE -> PREV = TEMP
• Step 11: EXIT
```



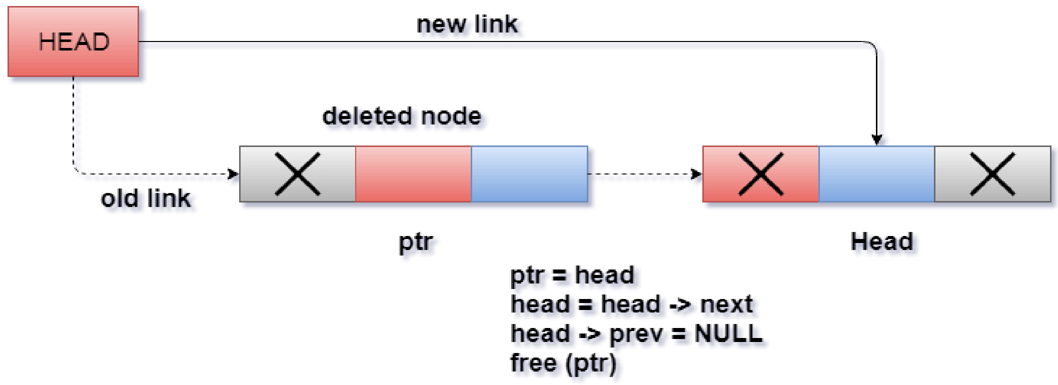
Insertion into doubly linked list at the end

Deletion

1)Deletes an element at the beginning of the list.

Algorithm

- **STEP 1:** IF HEAD = NULL
  WRITE UNDERFLOW
  GOTO STEP 6
- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → NEXT
- **STEP 4:** SET HEAD → PREV = NULL
- **STEP 5:** FREE PTR
- **STEP 6:** EXIT



Deletion in doubly linked list from beginning

2)Deletes an element at the given location of the list.

Algorithm

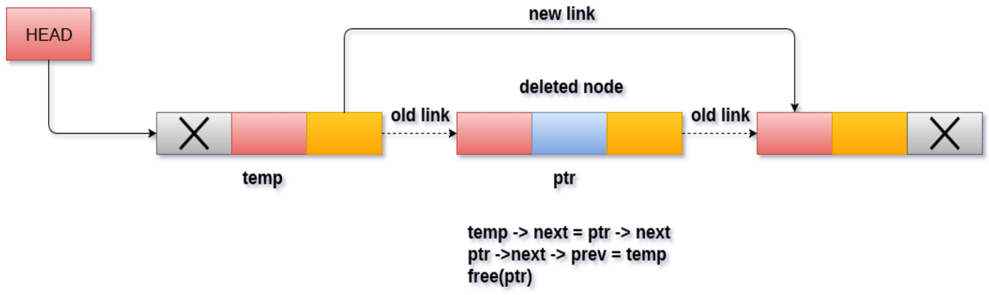
- **Step 1:** IF HEAD = NULL

```
  Write UNDERFLOW
 Go to Step 9
[END OF IF]
```

- **Step 2:** SET TEMP = HEAD
- **Step 3:** Repeat Step 4 while TEMP -> DATA != ITEM
- **Step 4:** SET TEMP = TEMP -> NEXT

 [END OF LOOP]

- **Step 5:** SET PTR = TEMP -> NEXT
- **Step 6:** SET TEMP -> NEXT = PTR -> NEXT
- **Step 7:** SET PTR -> NEXT -> PREV = TEMP
- **Step 8:** FREE PTR
- **Step 9:** EXIT



Deletion of a specified node in doubly linked list

3) Deletes an element at the end of the list.

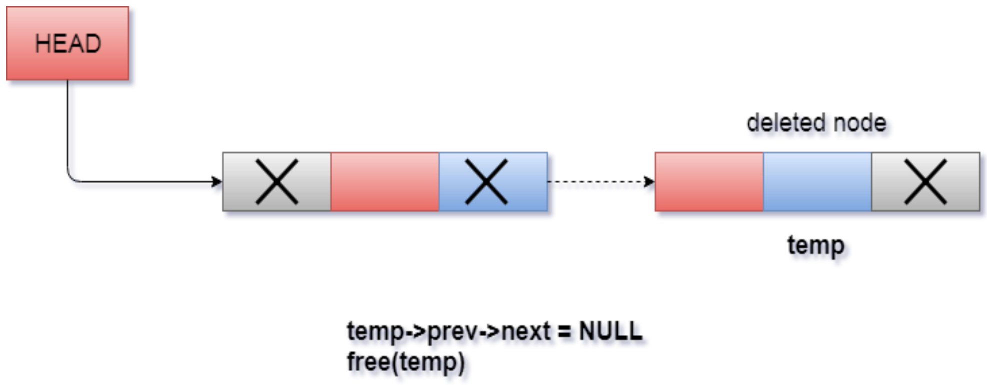
Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

Step 1: IF HEAD = NULL

```
    Write UNDERFLOW
    Go to Step 7
    [END OF IF]

Step 2: SET TEMP = HEAD

Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL

Step 4: SET TEMP = TEMP->NEXT

    [END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT
```



Deletion in doubly linked list at the end

Display − Displays the complete list.

- Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr. then, traverse through the list by using while loop. Keep shifting value of pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

Algorithm

- **Step 1:** IF HEAD == NULL

```
    WRITE "UNDERFLOW"
    GOTO STEP 6
    [END OF IF]
• Step 2: Set PTR = HEAD
• Step 3: Repeat step 4 and 5 while PTR != NULL
• Step 4: Write PTR → data
• Step 5: PTR = PTR → next
• Step 6: Exit
```

Search − Searches an element using the given key.

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

Algorithm

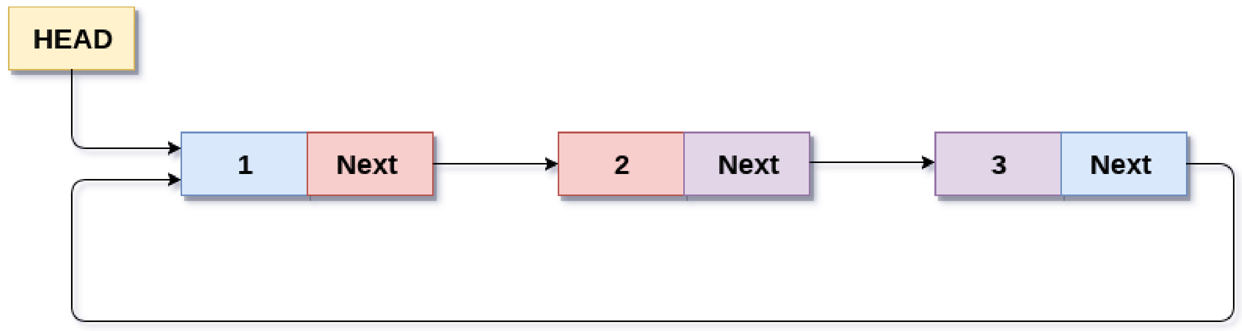
```
• Step 1: IF HEAD == NULL
    WRITE "UNDERFLOW"
    GOTO STEP 8
    [END OF IF]
• Step 2: Set PTR = HEAD
• Step 3: Set i = 0
• Step 4: Repeat step 5 to 7 while PTR != NULL
• Step 5: IF PTR → data = item
        return i
    [END OF IF]
• Step 6: i = i + 1
• Step 7: PTR = PTR → next
• Step 8: Exit
```

❖ Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



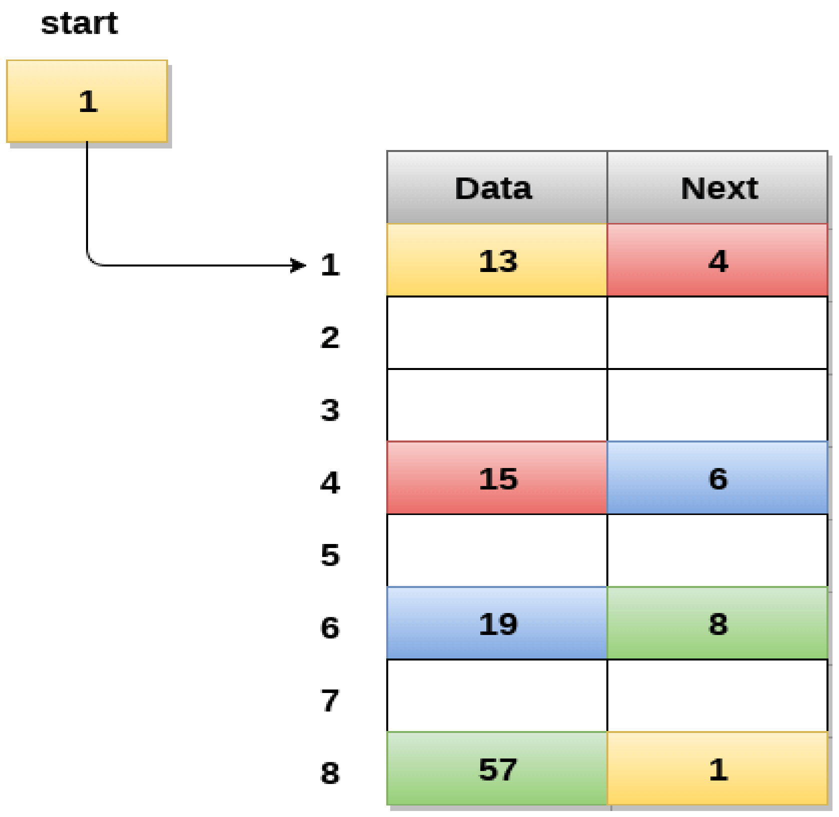
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

❖ **Memory Representation of circular linked list:**

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

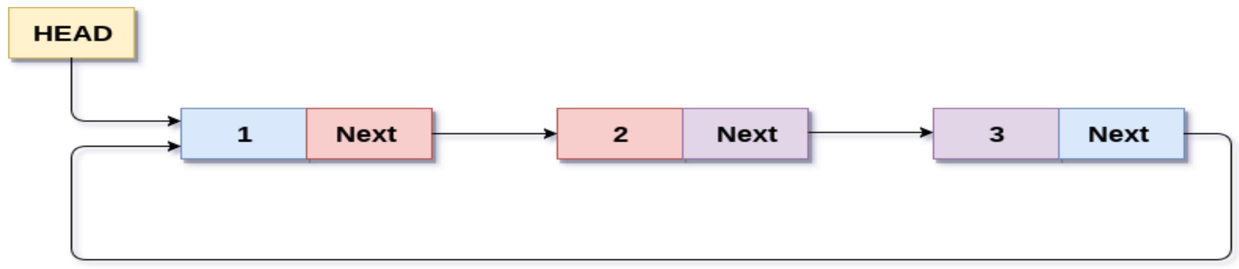
However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

❖ Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.
A circular doubly linked list is shown in the following figure.



Circular Singly Linked List

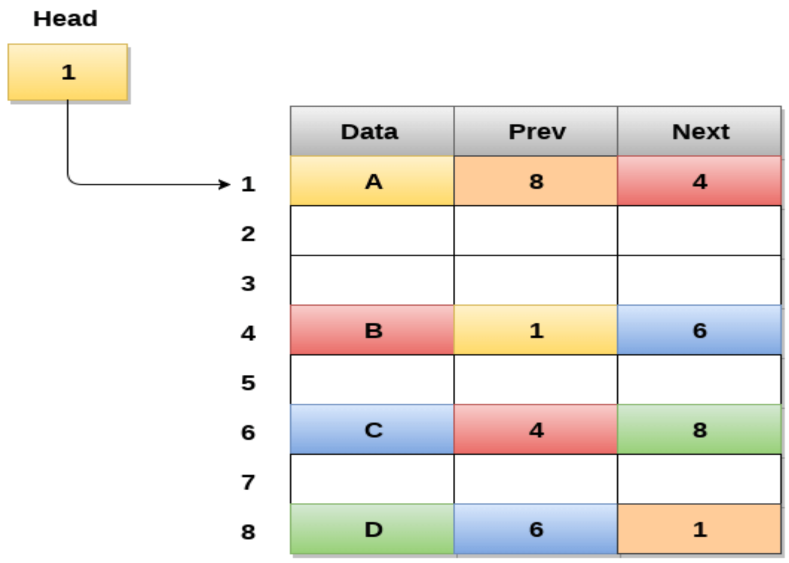
Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

❖ **Memory Management of Circular Doubly linked list**

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1. Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4. The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1. In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.



Memory Representation of a Circular Doubly linked list

❖ **Priority Queue**

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.
insert(item, priority): Inserts an item with given priority.
getHighestPriority (): Returns the highest priority item.
DeleteHighestPriority (): Removes the highest priority item.

❖ **Dynamic storage management**

Resources are always a premium. We have strived to achieve better utilization of resources at all times; that is the premise of our progress. Related to this pursuit, is the concept of memory allocation. Memory has to be allocated to the variables that we create, so that actual variables can be brought to existence. Now there is a constraint as how we think it happens, and how it actually happens.

❖ **Garbage collection**

Garbage collection (GC) is a dynamic approach to automatic memory management and heap allocation that processes and identifies dead memory blocks and

reallocates storage for reuse. The primary purpose of garbage collection is to reduce memory leaks.
GC implementation requires three primary approaches, as follows:
Mark-and-sweep - In process when memory runs out, the GC locates all accessible memory and then reclaims available memory.
Reference counting - Allocated objects contain a reference count of the referencing number. When the memory count is zero, the object is garbage and is then destroyed. The freed memory returns to the memory heap.
Copy collection - There are two memory partitions. If the first partition is full, the GC locates all accessible data structures and copies them to the second partition, compacting memory after GC process and allowing continuous free memory.

❖ Tree

In computer science, a **tree** is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.
A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

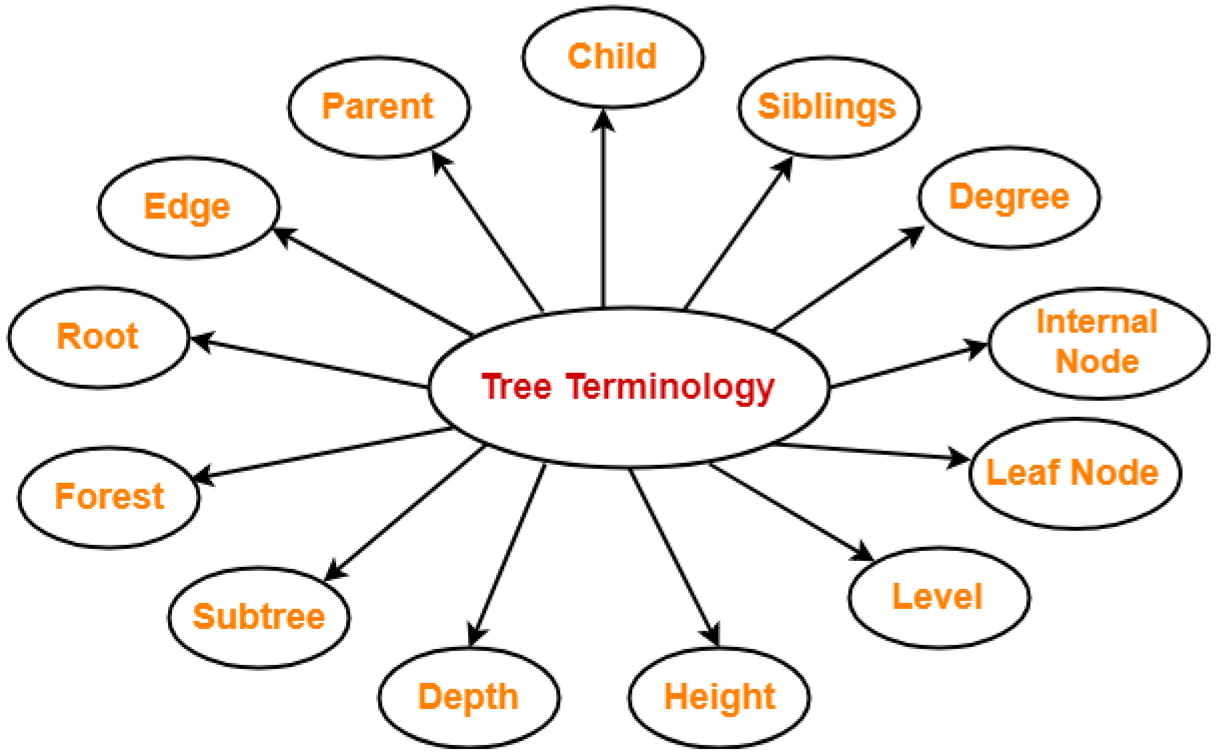
Properties-

The important properties of tree data structure are-

- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has exactly (n-1) edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and (n-1) edges is a tree.

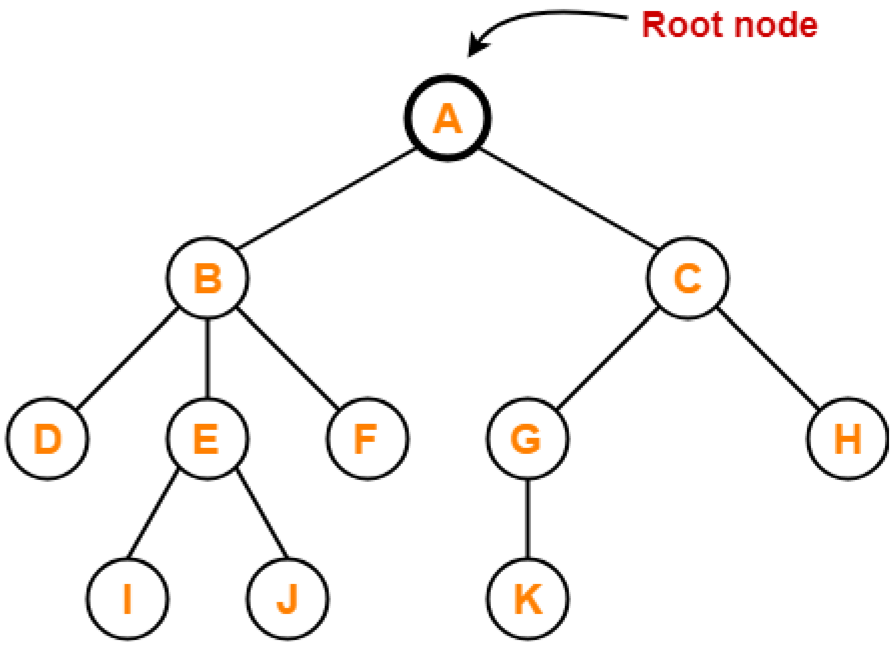
Tree Terminology-

The important terms related to tree data structure are-



1. Root-

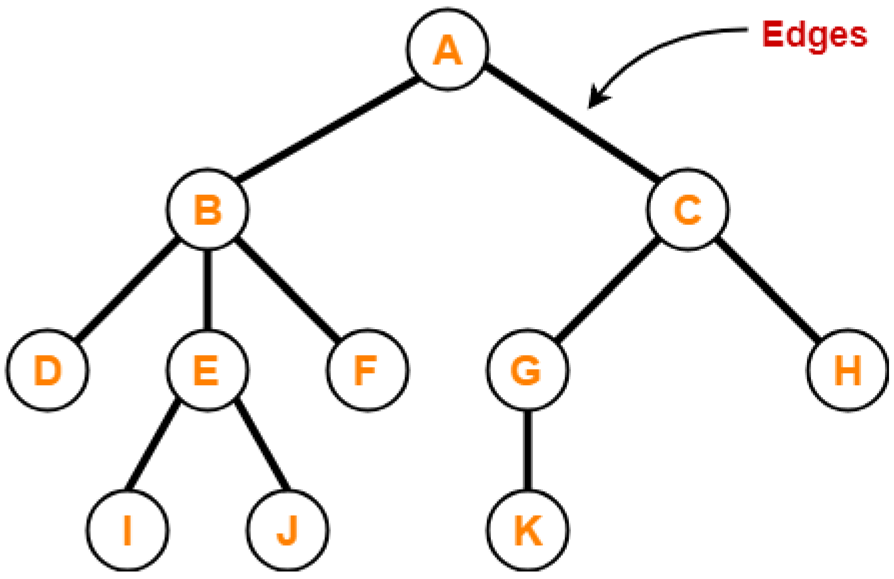
- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.



2. Edge-

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.

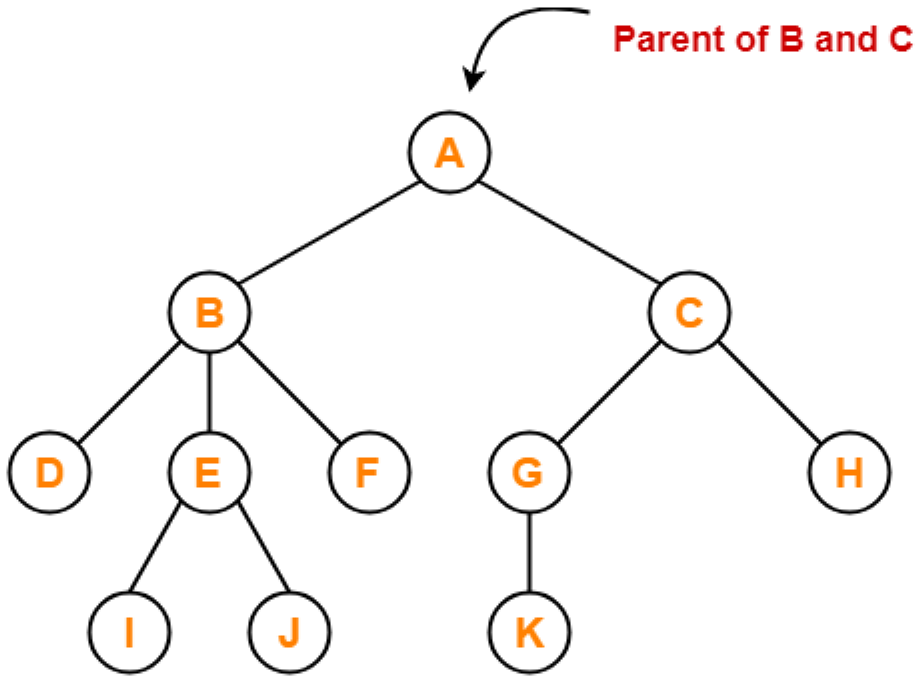
Example-



3. Parent-

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

Example-



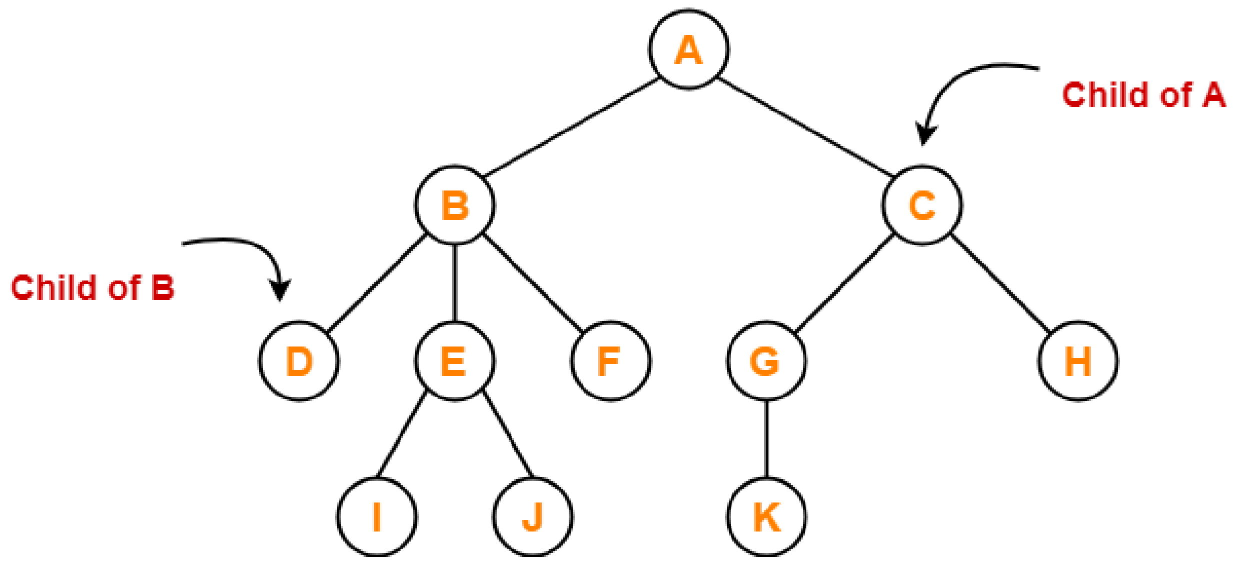
Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

4. Child-

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

<u>**Example-**</u>



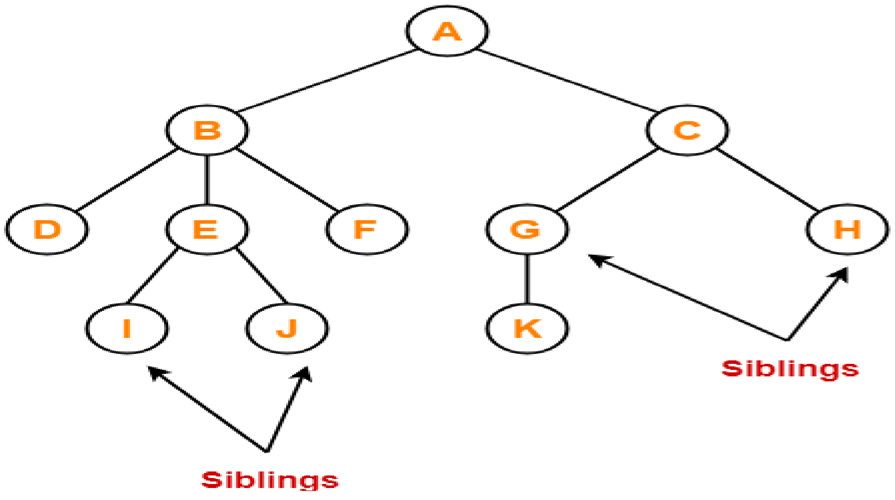
Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

5. Siblings-

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

__**Example-**__



- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

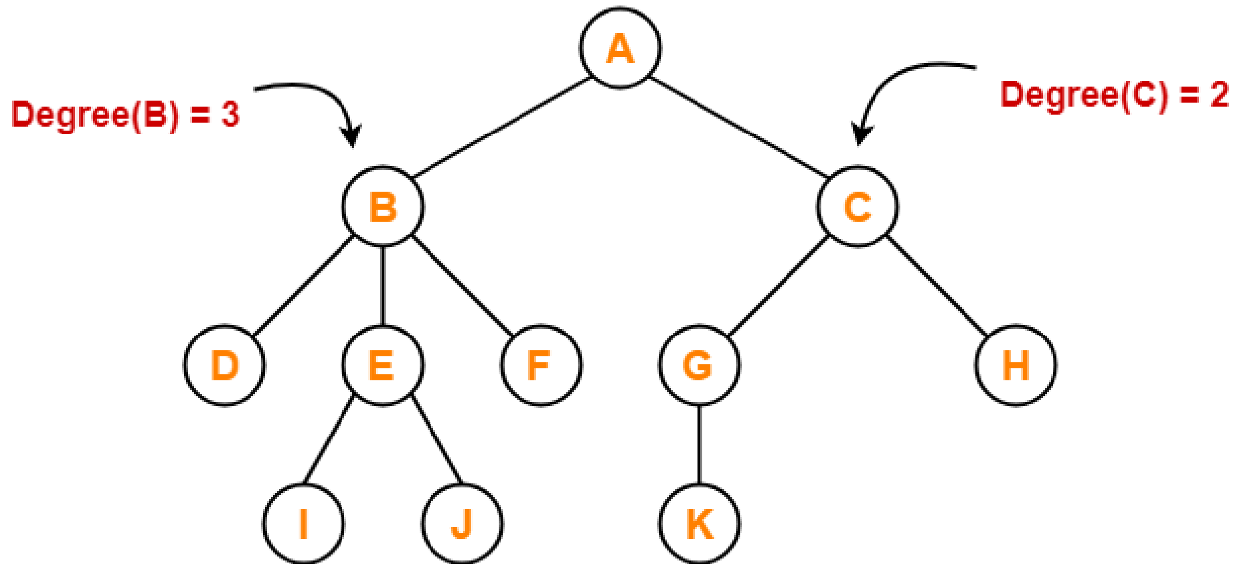
6. Degree-

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

Example-

Here,

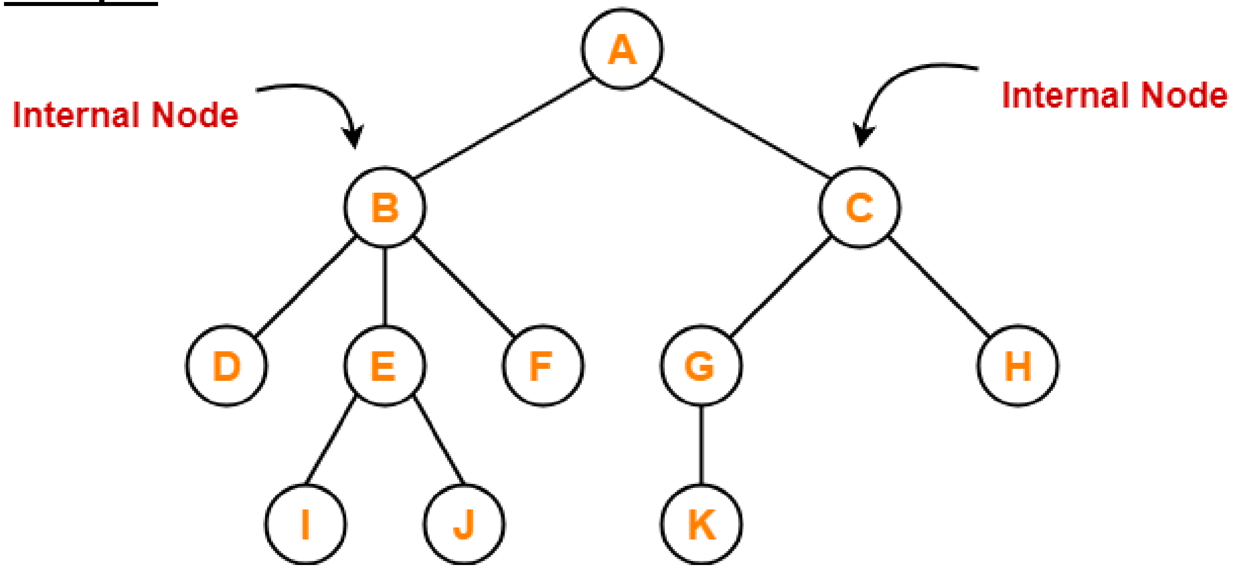
- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0



7. Internal Node-

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

Example-

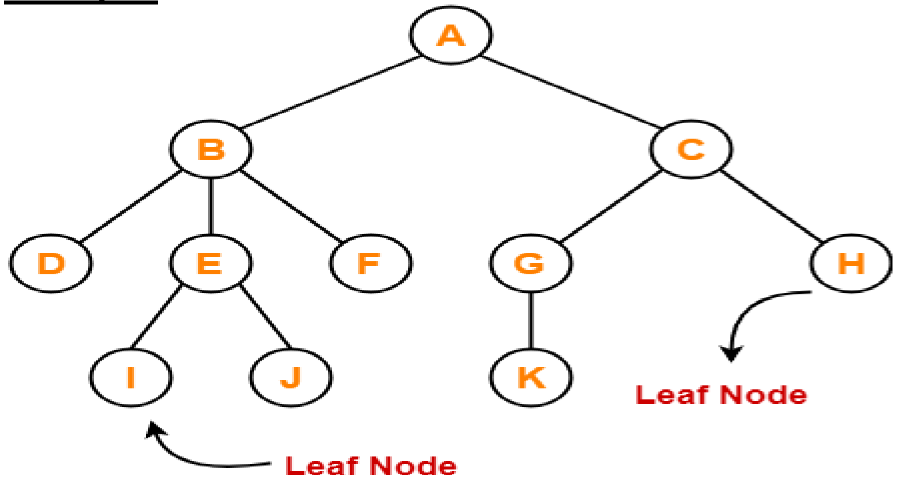


Here, nodes A, B, C, E and G are internal nodes.

8. Leaf Node-

- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.

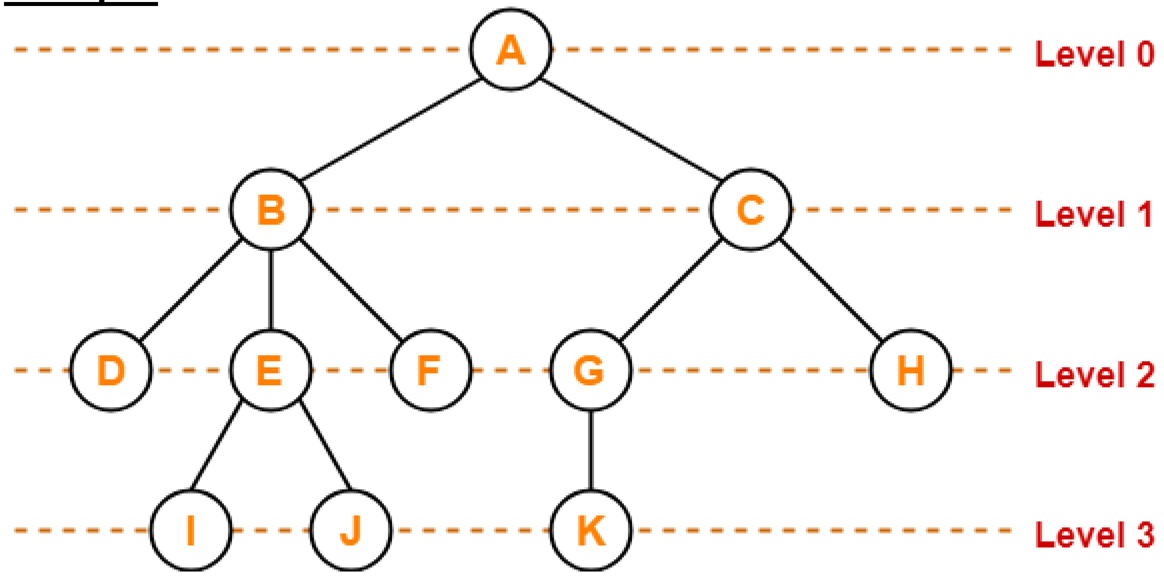
Example-



9. Level-

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

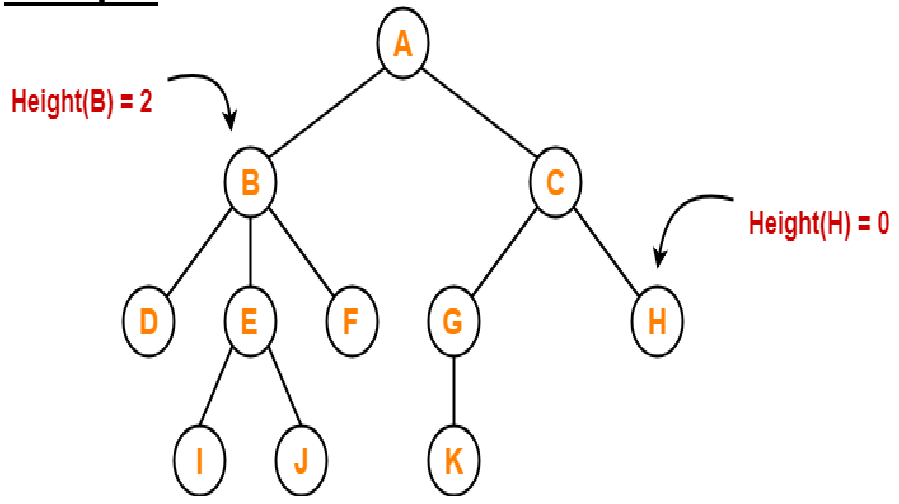
Example-



10. Height-

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0

Example-



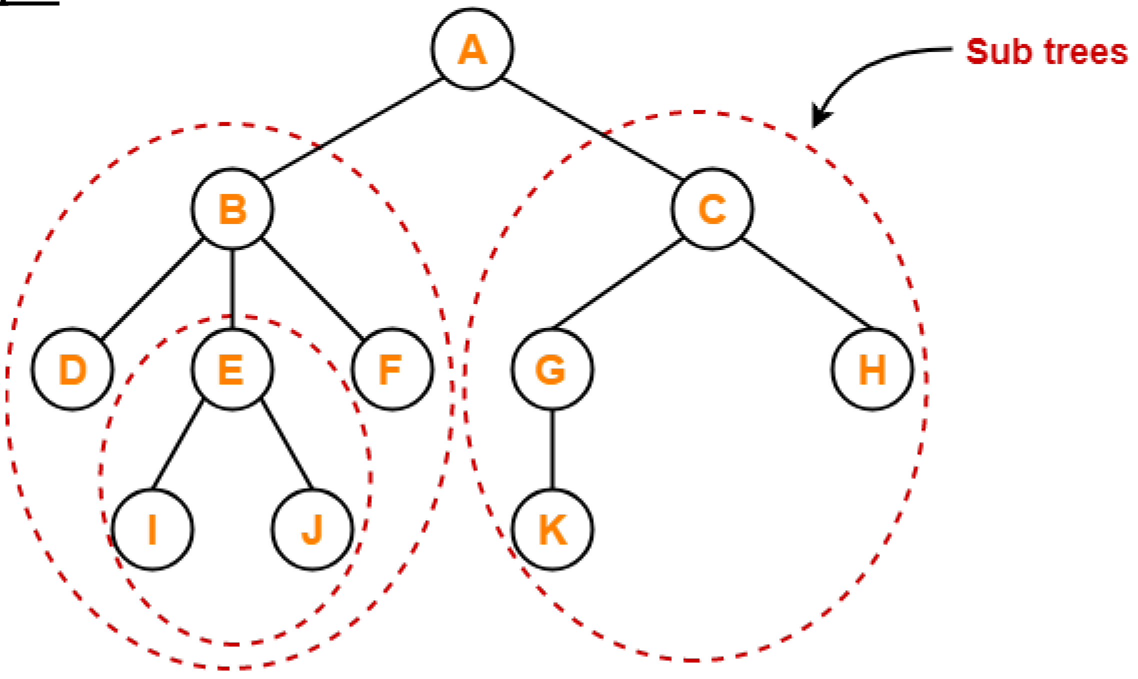
Here,

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

11. Subtree-

- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.

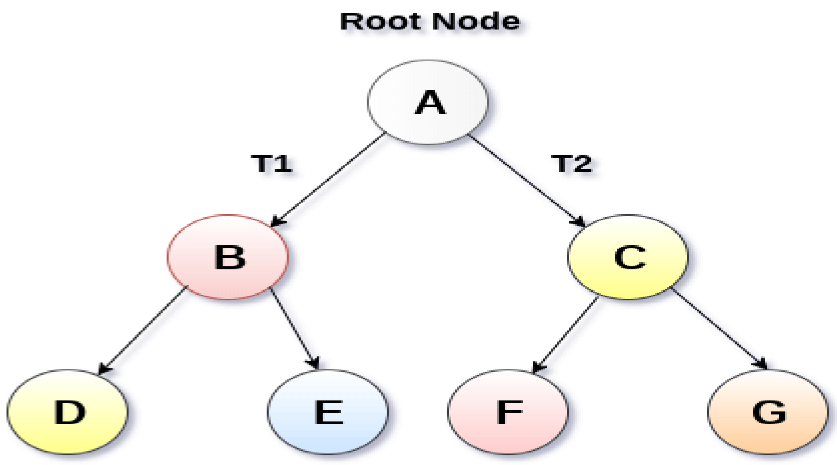
Example-



❖ Binary Tree

We extend the concept of linked data structures to structure containing nodes with more than one self-referenced field. A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.
Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.
More tree terminology:

- The depth of a node is the number of edges from the root to the node.
- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.
- A full binary tree.is a binary tree in which each node has exactly zero or two children.
- A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.



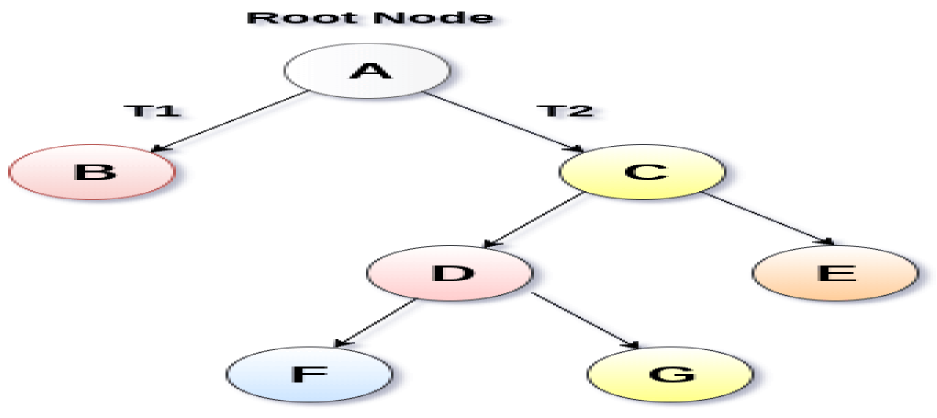
Binary Tree

Types of Binary Tree

1. Strictly Binary Tree

In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have (2n - 1) nodes.

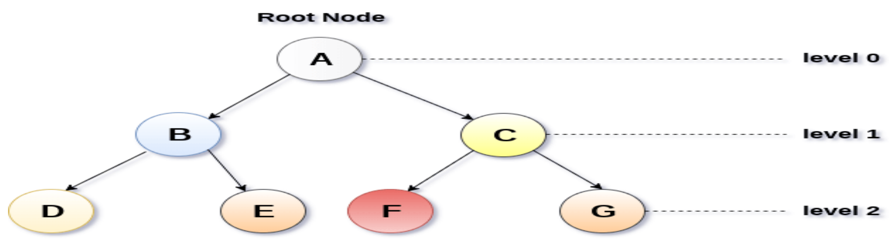
A strictly binary tree is shown in the following figure.



Strictly Binary Tree

2. Complete Binary Tree

A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d. A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d. The total number of nodes in a complete binary tree with depth d is 2^{d+1}-1 where leaf nodes are 2^d while non-leaf nodes are 2^d-1.



Complete Binary Tree

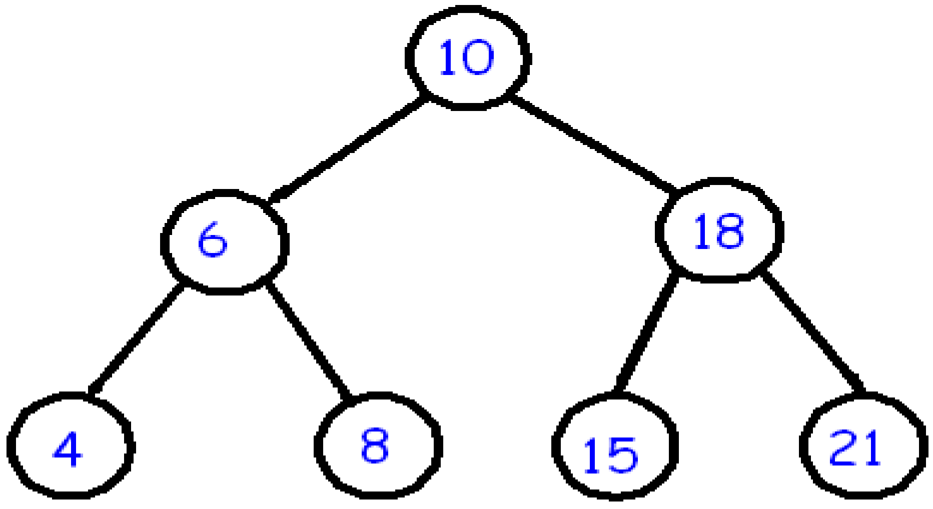
❖ Binary Search Trees

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retriving.

A BST is a binary tree where nodes are ordered in the following way:

- each node contains one key (also known as data)
- the keys in the left subtree are less then the key in its parent node, in short L < P;
- the keys in the right subtree are greater the key in its parent node, in short P < R;
- duplicate keys are not allowed.

In the following tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10. Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes:



Advantages of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2 n$) time. In worst case, the time it takes to search an element is 0(n).

3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

❖ Threaded Binary Tree

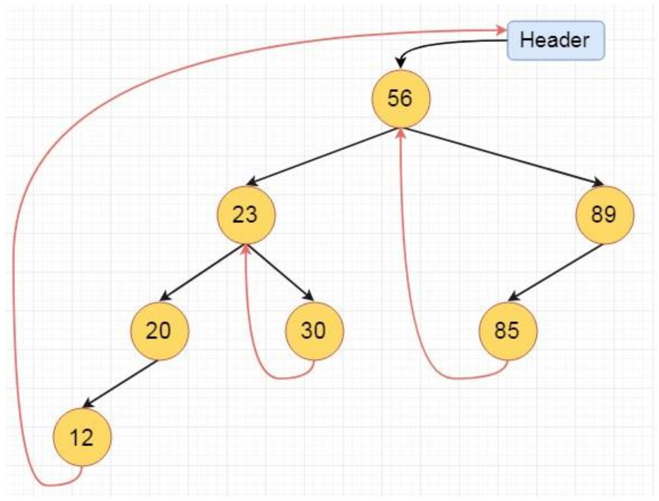
Threaded binary tree data structure. We know that the binary tree nodes may have at most two children. But if they have only one children, or no children, the link part in the linked list representation remains null. Using threaded binary tree representation, we can reuse that empty links by making some threads.

If one node has some vacant left or right child area, that will be used as thread. There are two types of threaded binary tree. The **single threaded tree or fully threaded binary tree**. In single threaded mode, there are another two variations. Left threaded and right threaded.

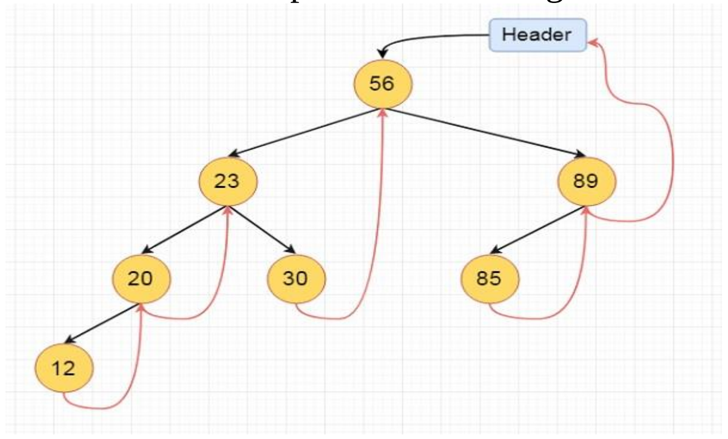
In the left threaded mode if some node has no left child, then the left pointer will point to its inorder predecessor, similarly in the right threaded mode if some node has no right child, then the right pointer will point to its inorder successor. In both cases, if no successor or predecessor is present, then it will point to header node.

For fully threaded binary tree, each node has five fields. Three fields like normal binary tree node, another two fields to store Boolean value to denote whether link of that side is actual link or thread.

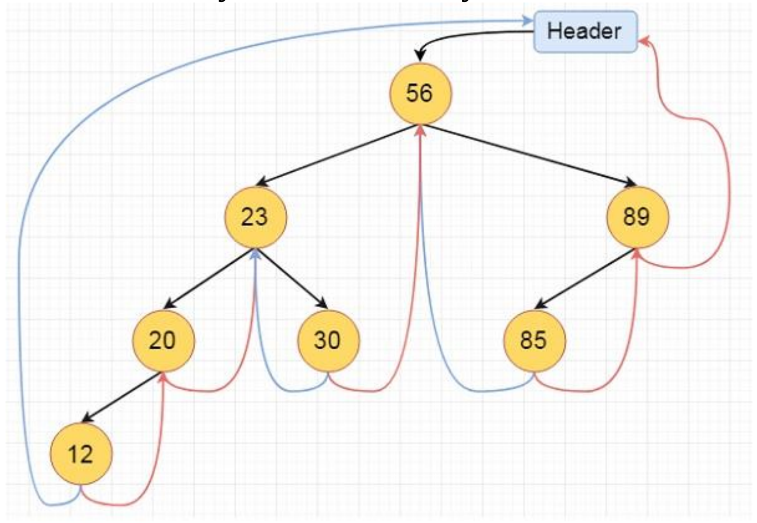
Left Thread Flag	Left Link	Data	Right Link	Right Thread Flag



These are the examples of left and right threaded tree



This is the fully threaded binary tree



❖ AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

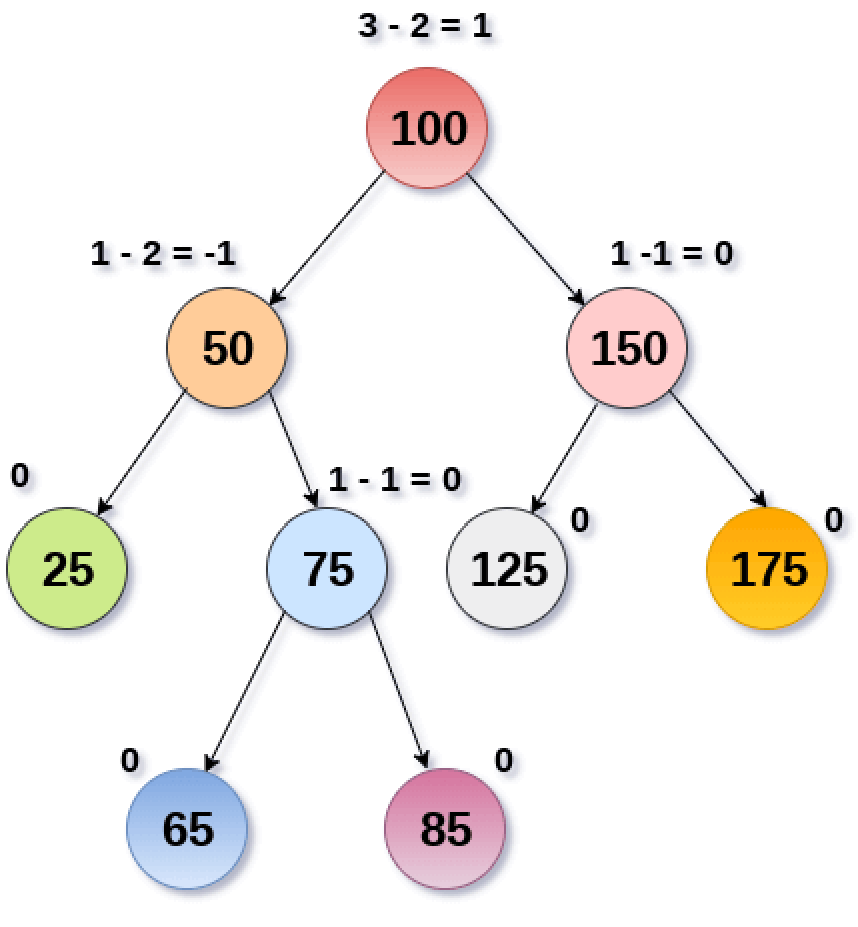
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Why AVL Tree?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

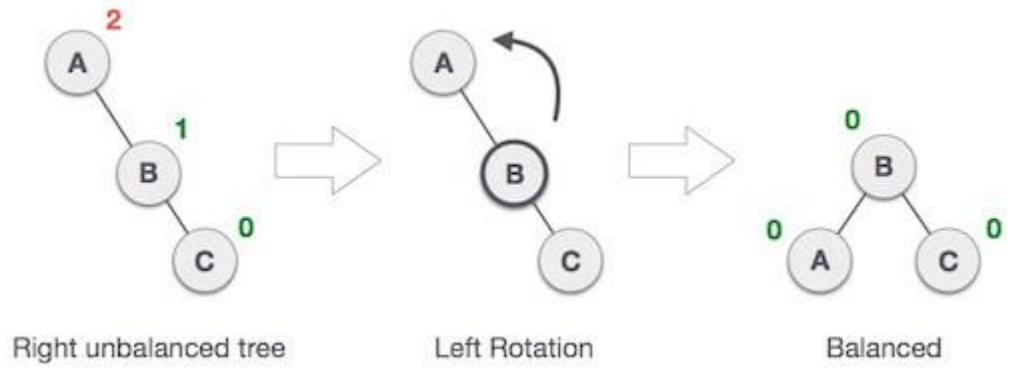
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

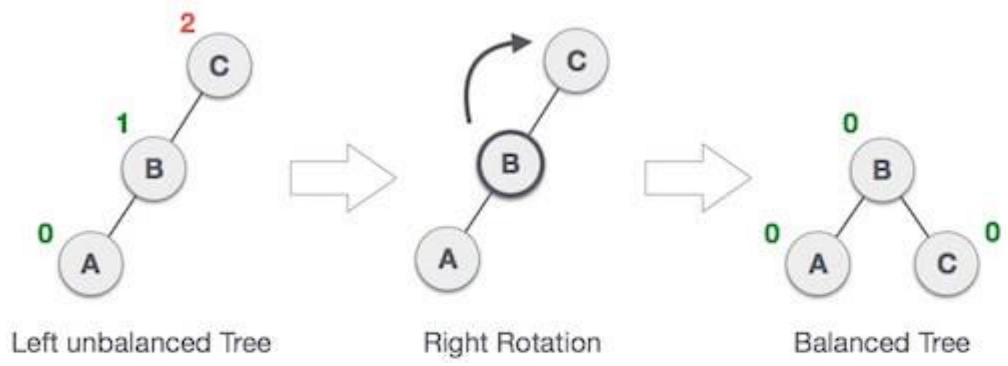
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



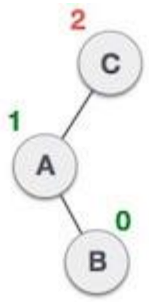
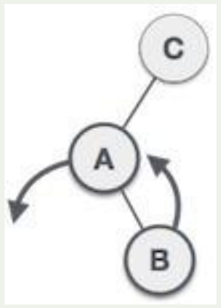
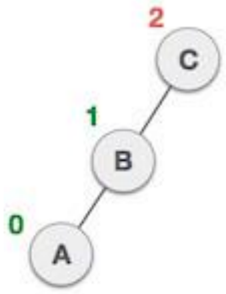
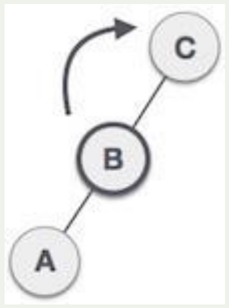
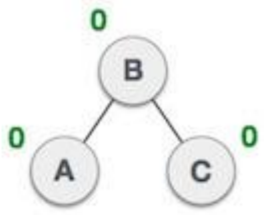
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

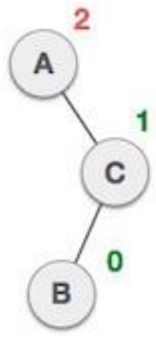
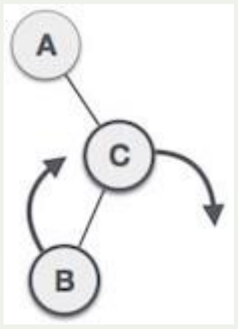
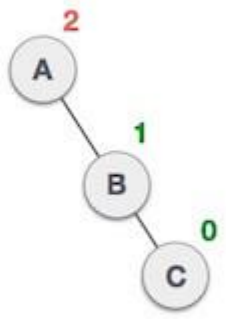
Let us understand each and every step very clearly:

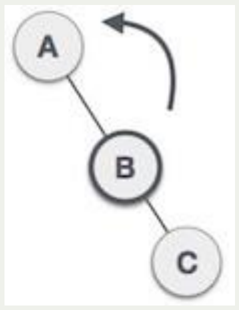
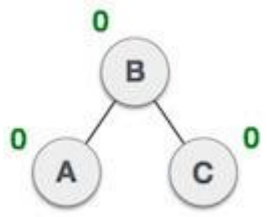
State	Action

	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

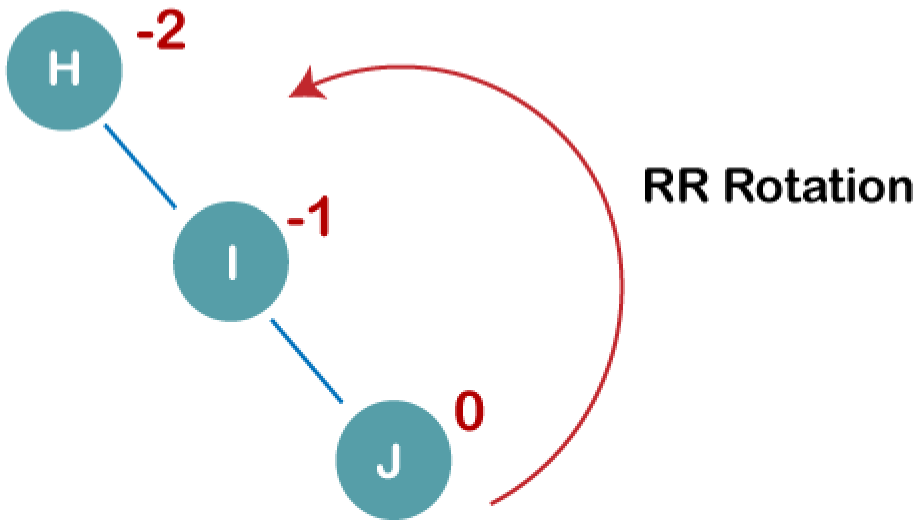
State	Action
	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.

	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

Construct an AVL tree having the following elements

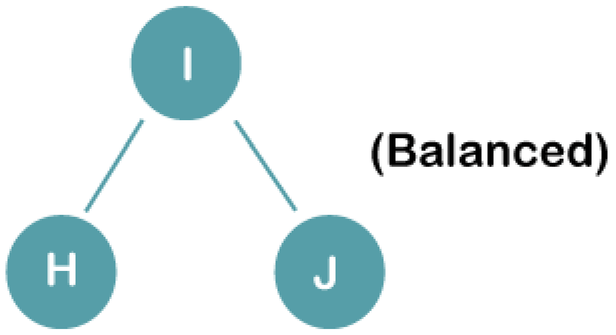
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

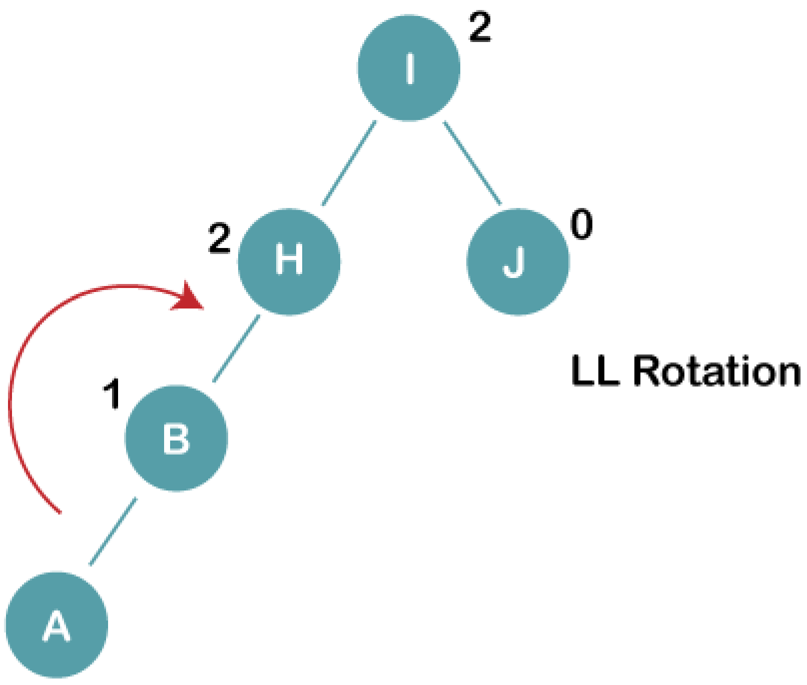


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

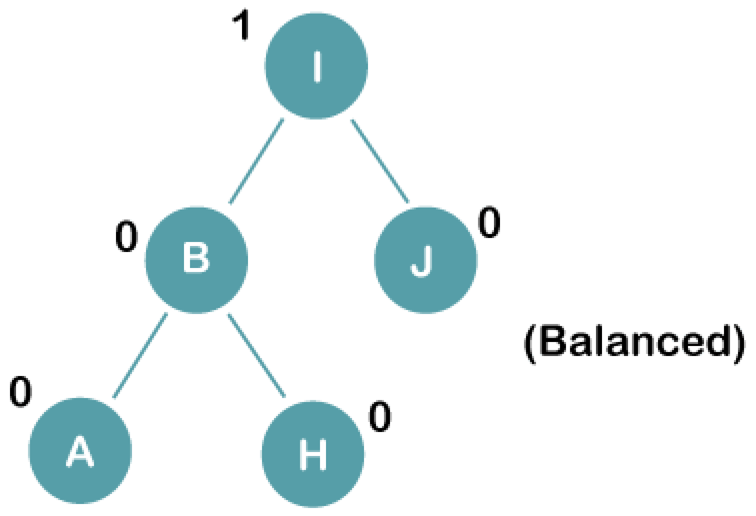


2. Insert B, A

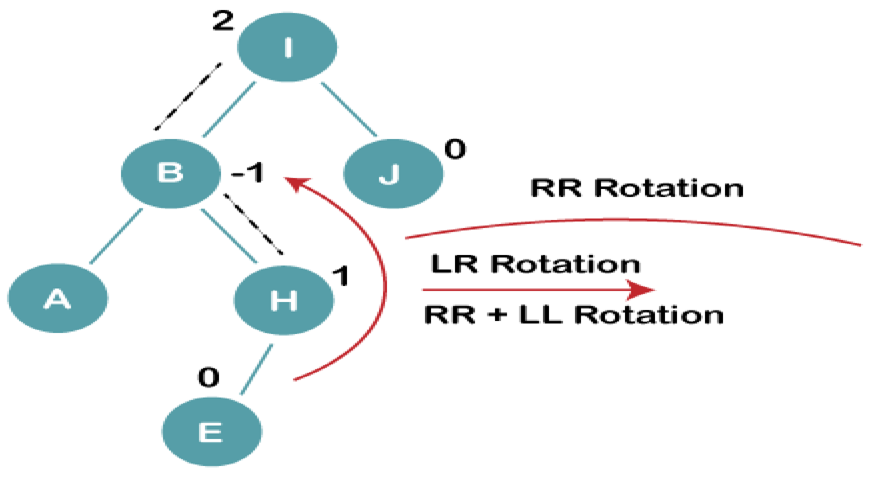


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



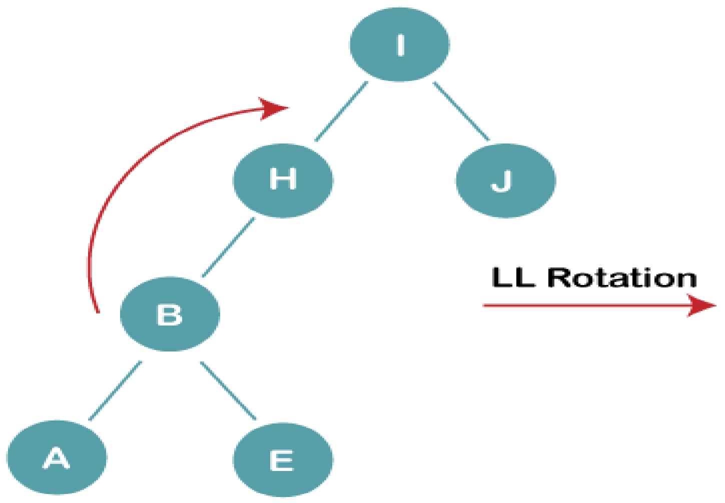
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

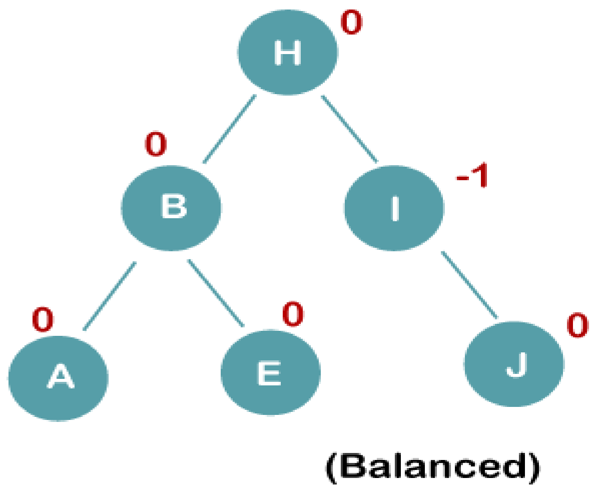
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

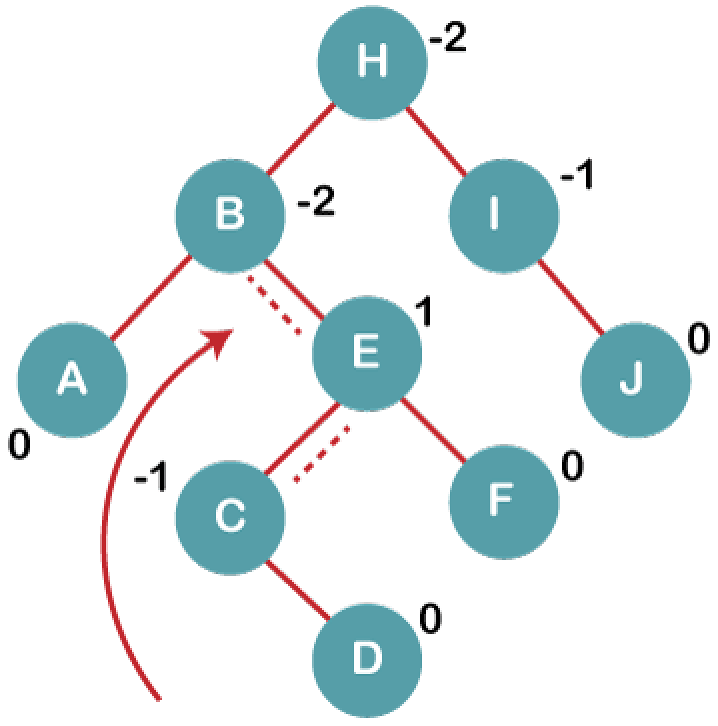


3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



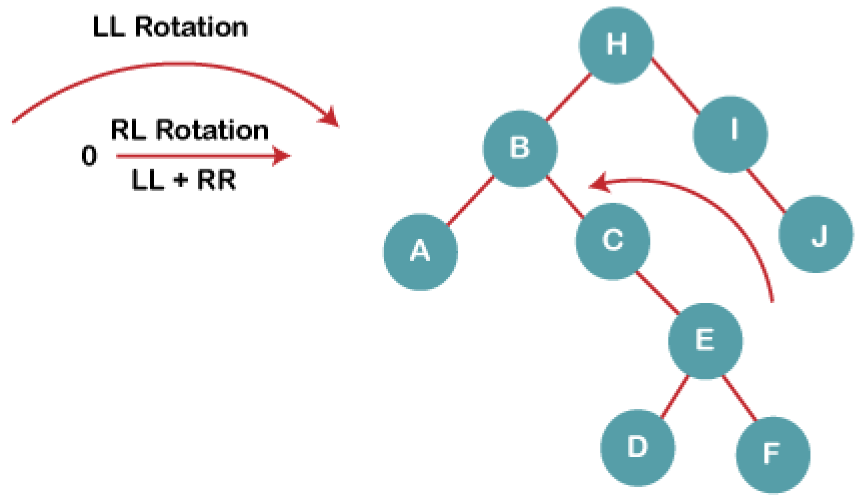
4. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

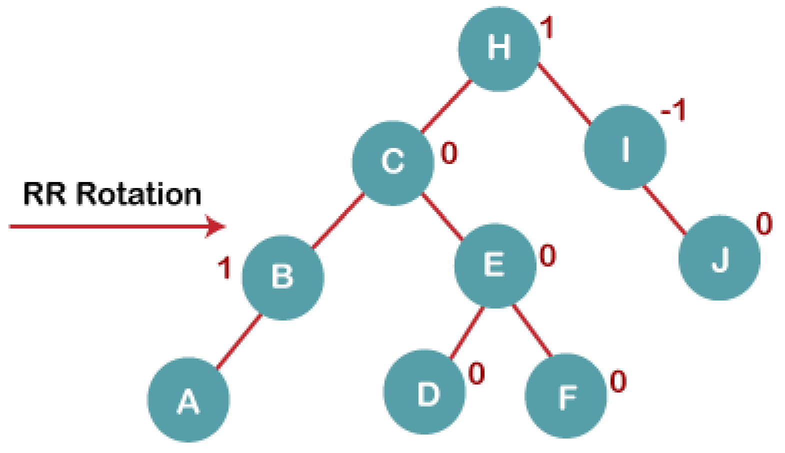
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:



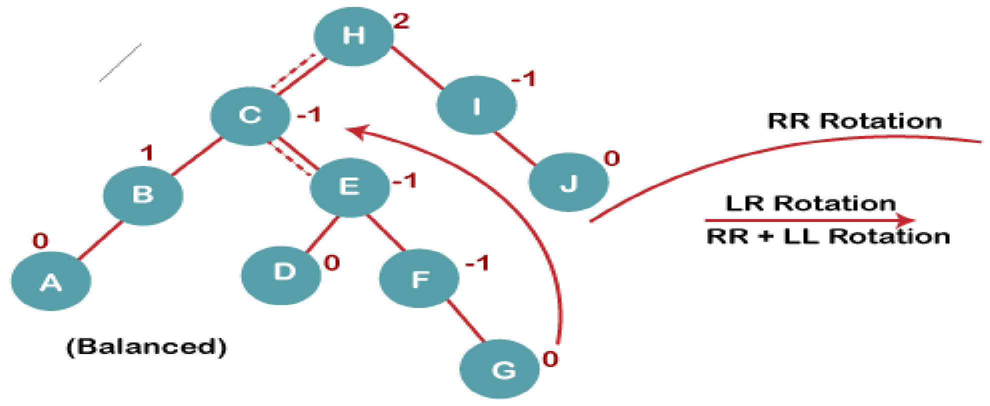
4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



(Balanced)

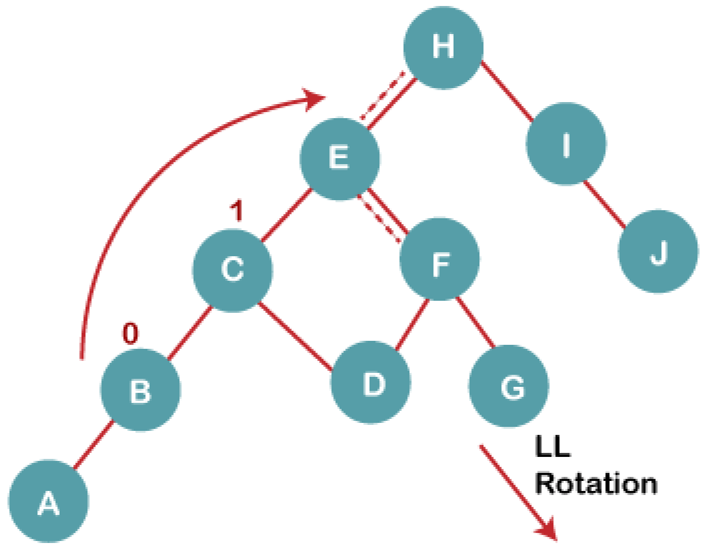
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

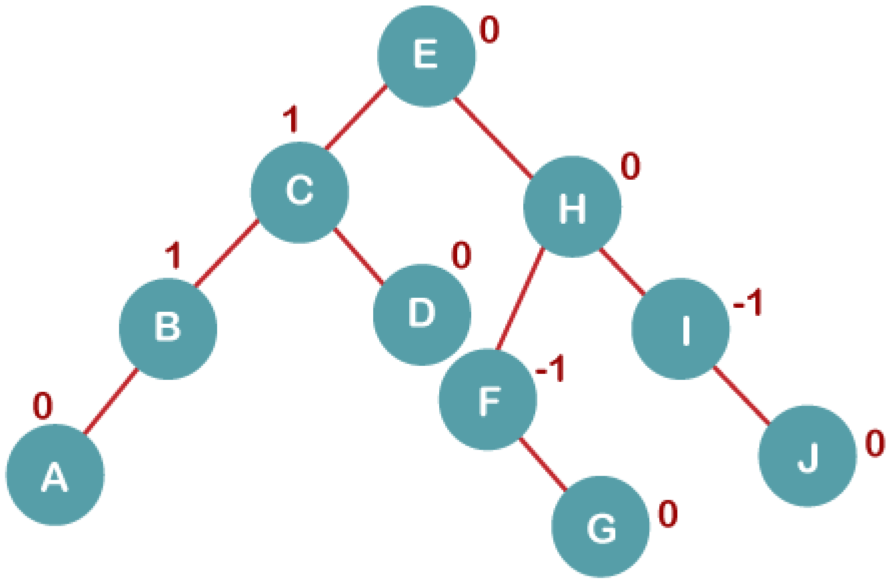
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:



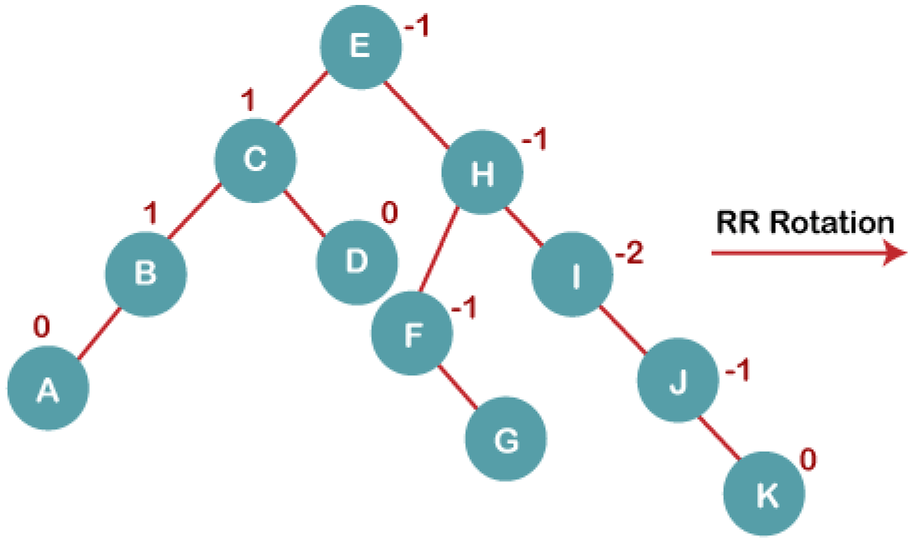
5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:



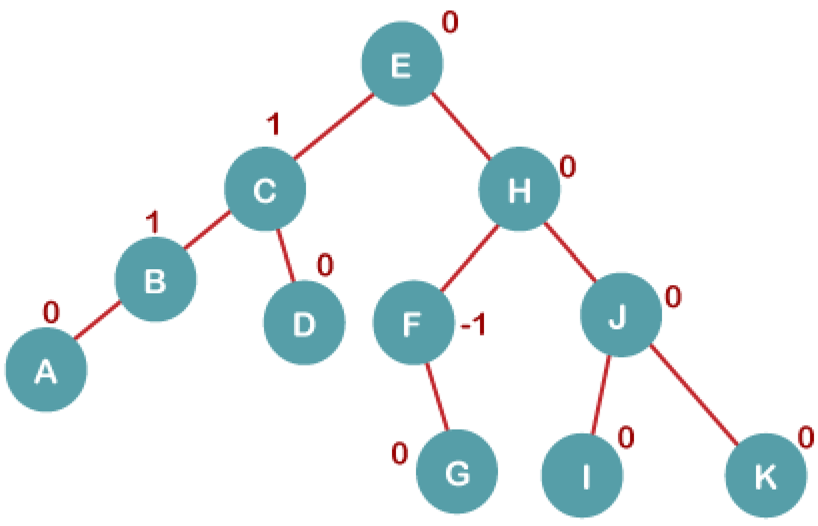
(Balanced)

6. Insert K



On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

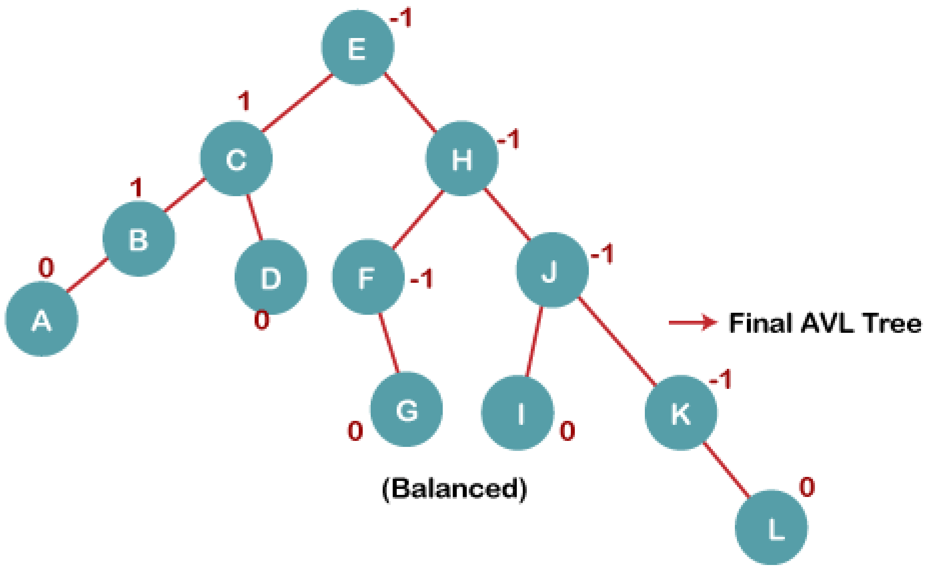
The resultant balanced tree after RR rotation is:



(Balanced)

7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



(Balanced)

❖ B Tree

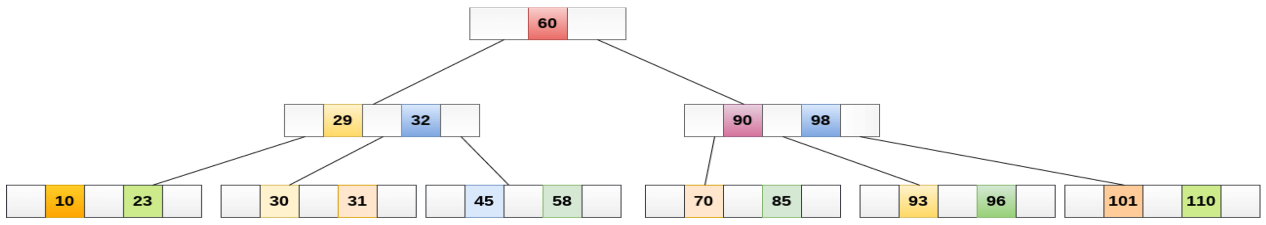
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

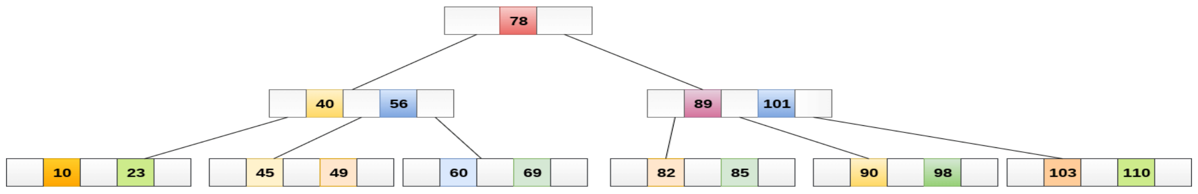
Operations

Searching:

Searching in B Trees is similar to that in Binary search tree. **For example**, if we search for an item 49 in the following B Tree. The process will something like following:

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



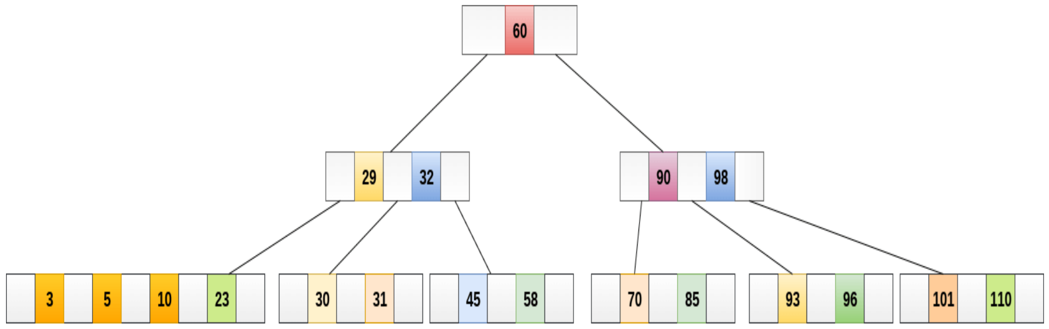
Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

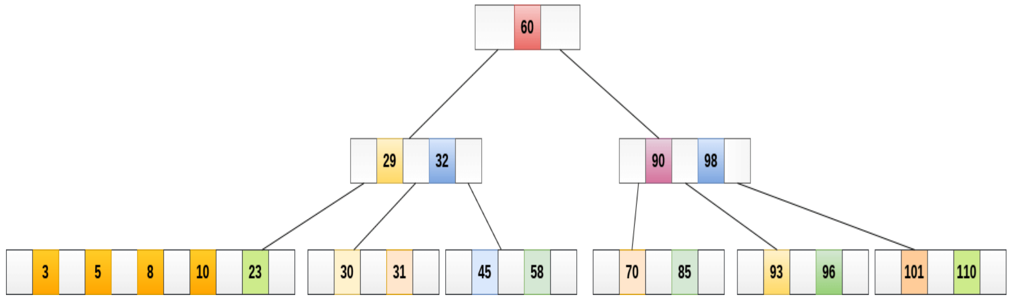
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   - Insert the new element in the increasing order of elements.
   - Split the node into the two nodes at the median.
   - Push the median element upto its parent node.
   - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

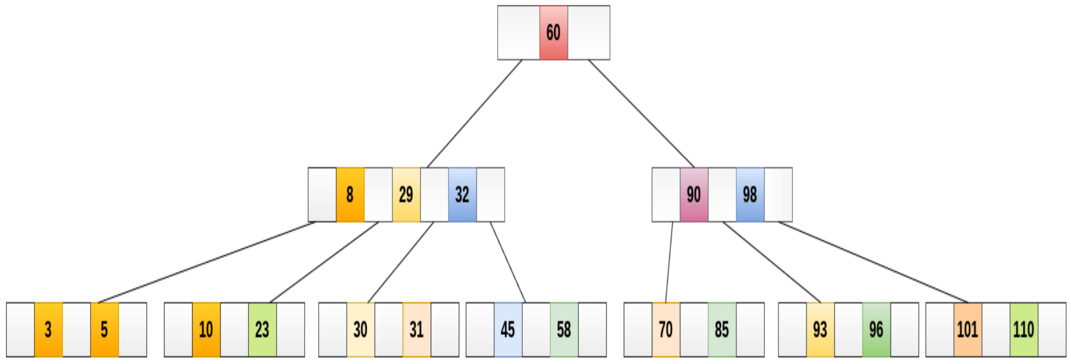
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

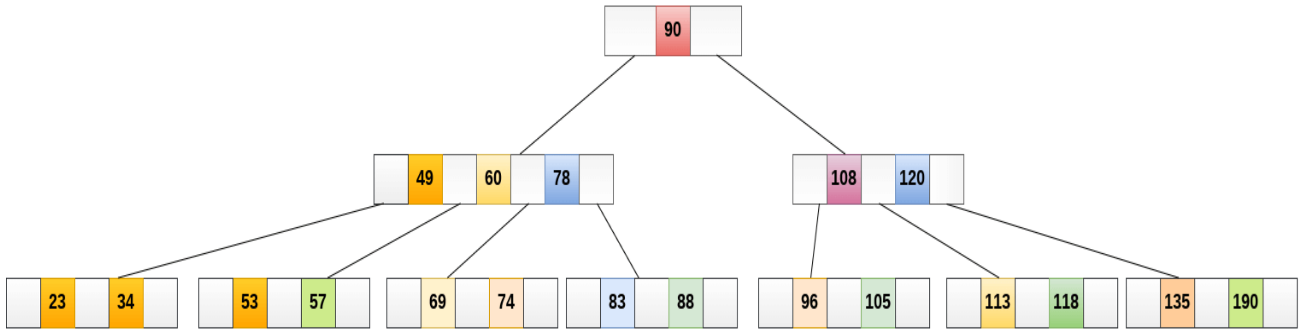
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

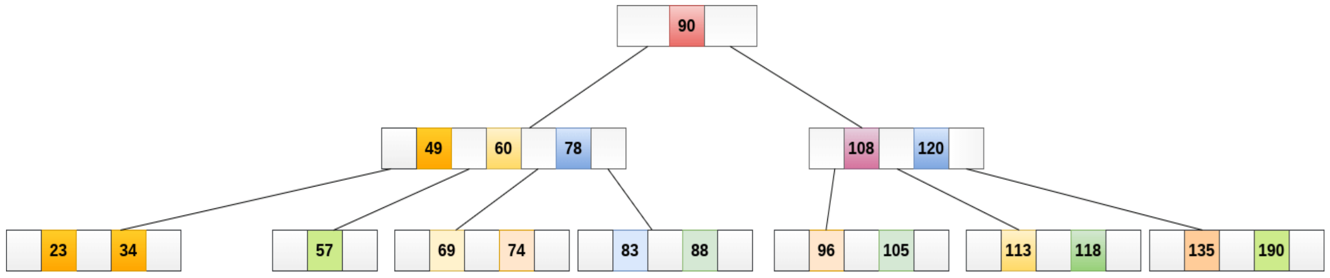
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

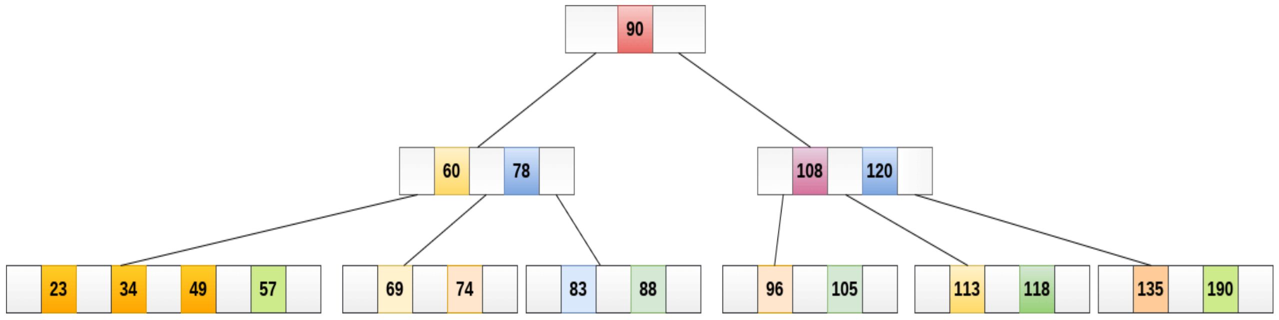


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

❖ B+ Tree

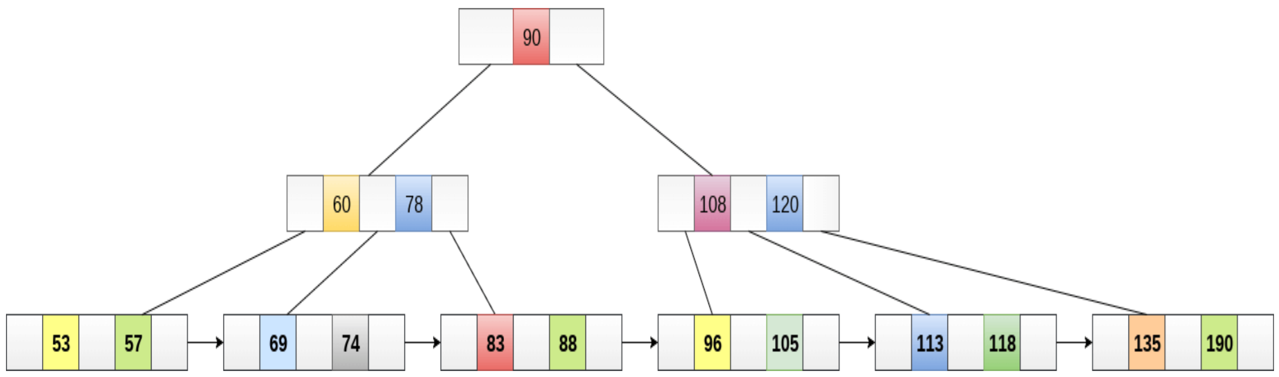
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

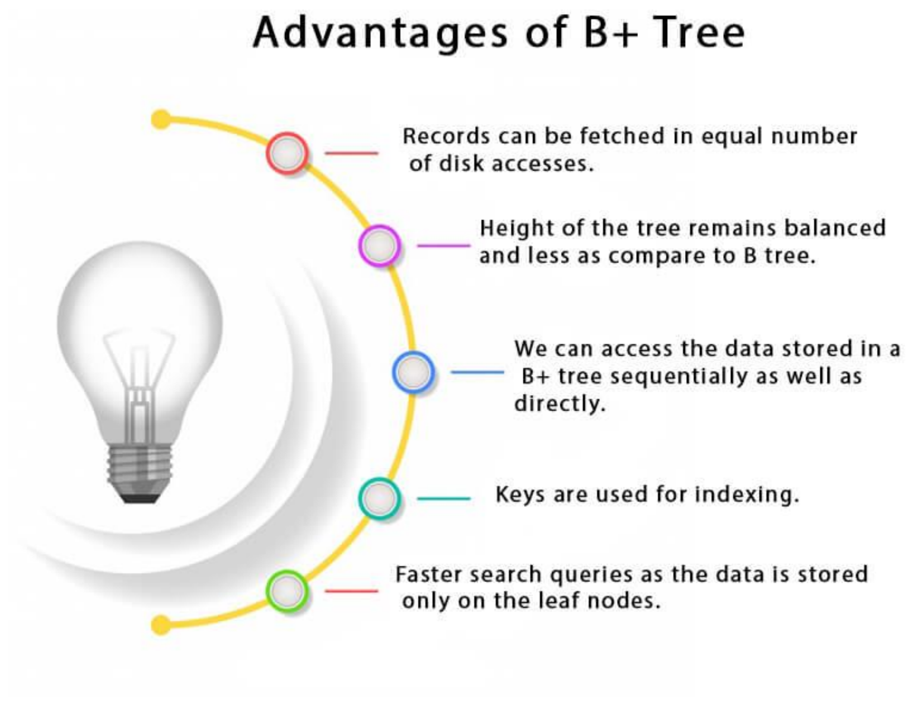
B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.


Advantages of B+ Tree
Records can be fetched in equal number of disk accesses.
Height of the tree remains balanced and less as compare to B tree.
We can access the data stored in a B+ tree sequentially as well as directly.
Keys are used for indexing.
Faster search queries as the data is stored only on the leaf nodes.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the

		leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Insertion in B+ Tree

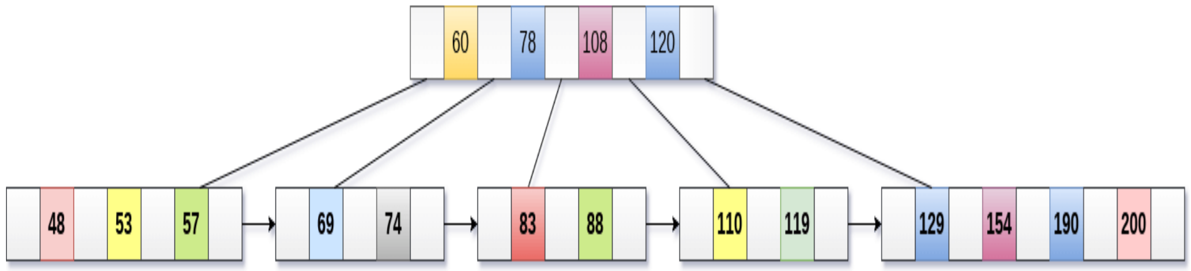
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

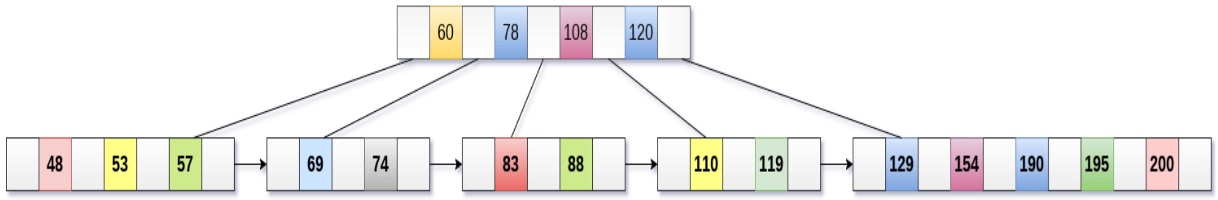
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example:

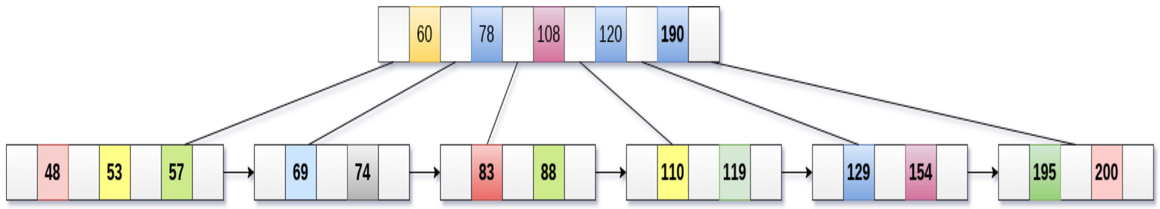
Insert the value 195 into the B+ tree of order 5 shown in the following figure.



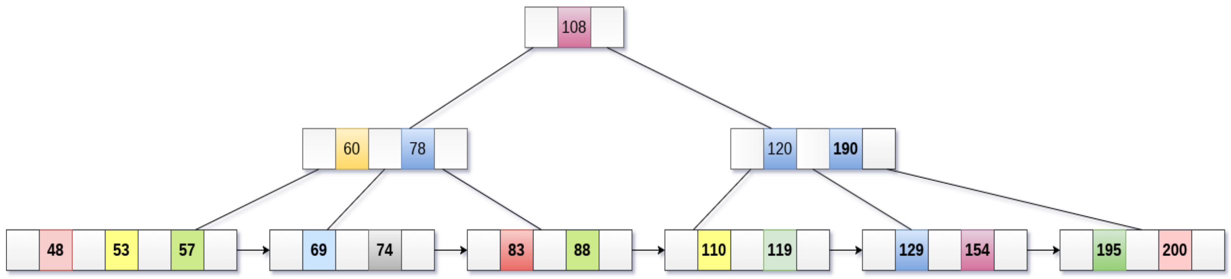
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

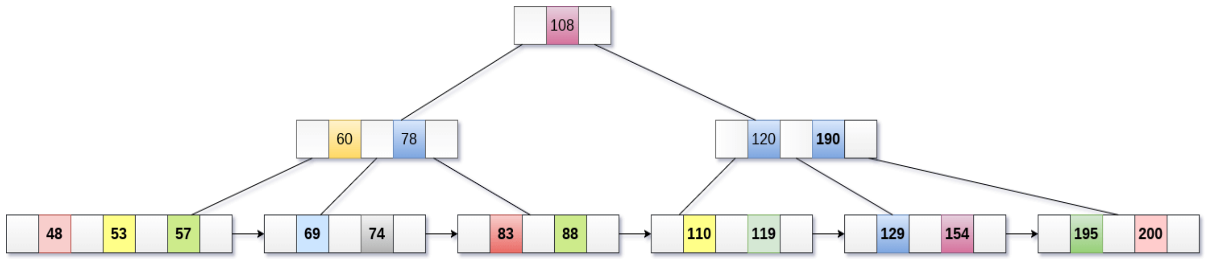
Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

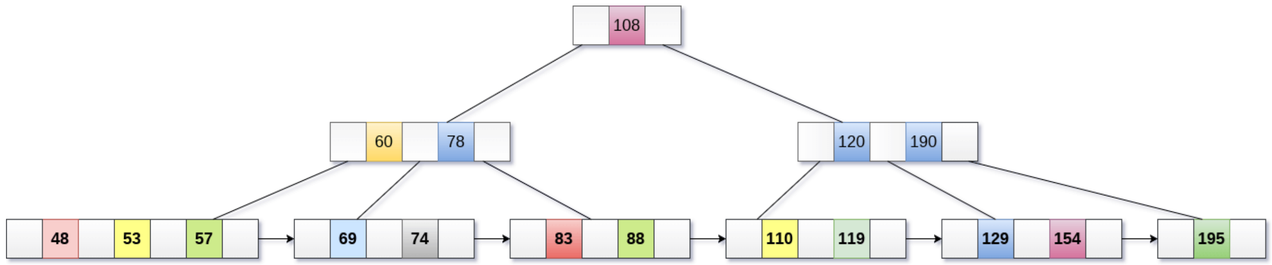
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

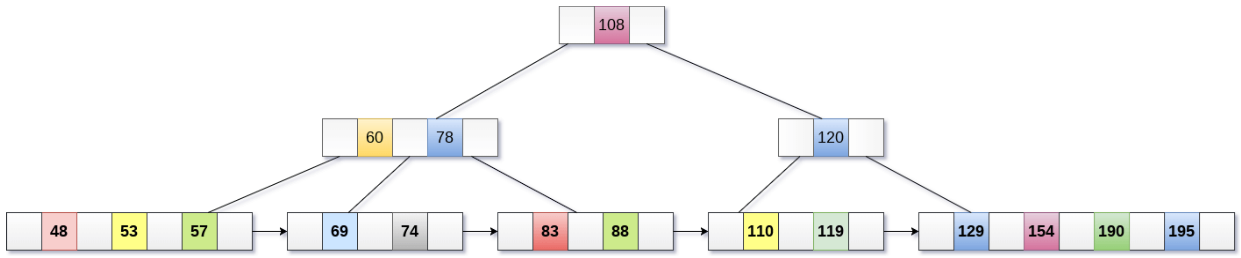
Delete the key 200 from the B+ Tree shown in the following figure.



200 is present in the right sub-tree of 190, after 195. delete it.

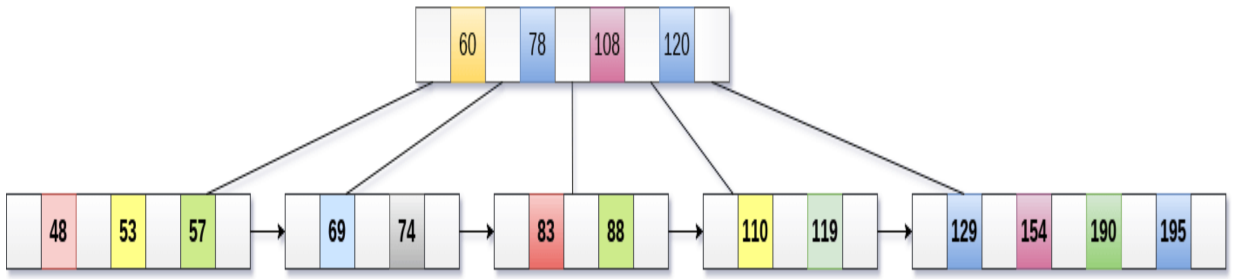


Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

Now, the height of B+ tree will be decreased by 1.



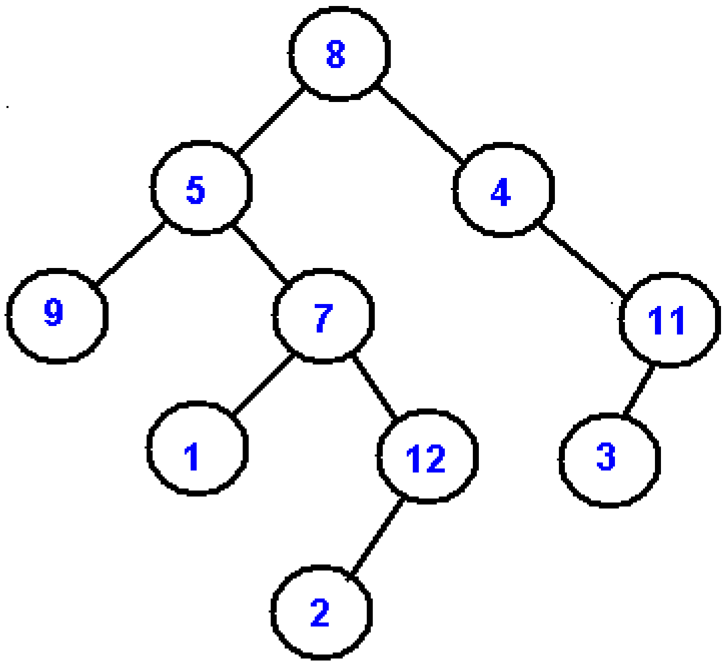
❖ Tree Traversals

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds

- depth-first traversal
- breadth-first traversal

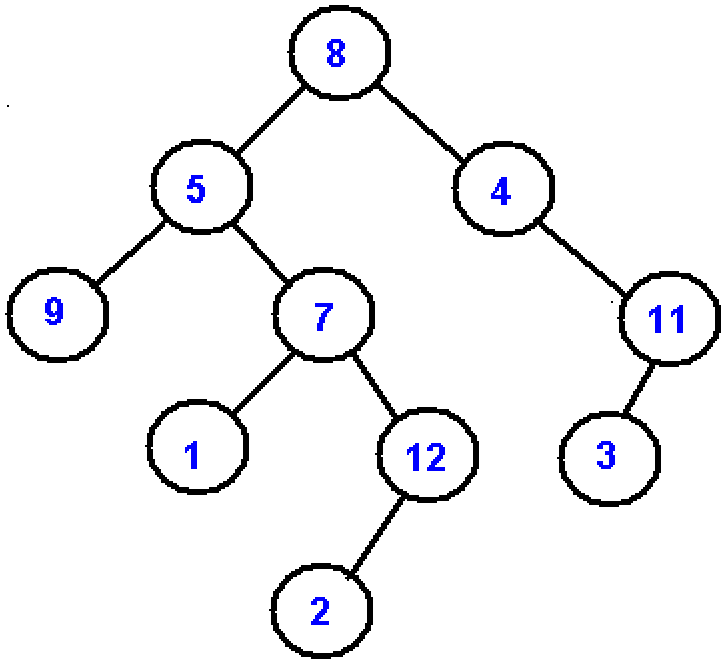
There are three different types of depth-first traversals, :

PreOrder traversal - visit the parent first and then left and right children;



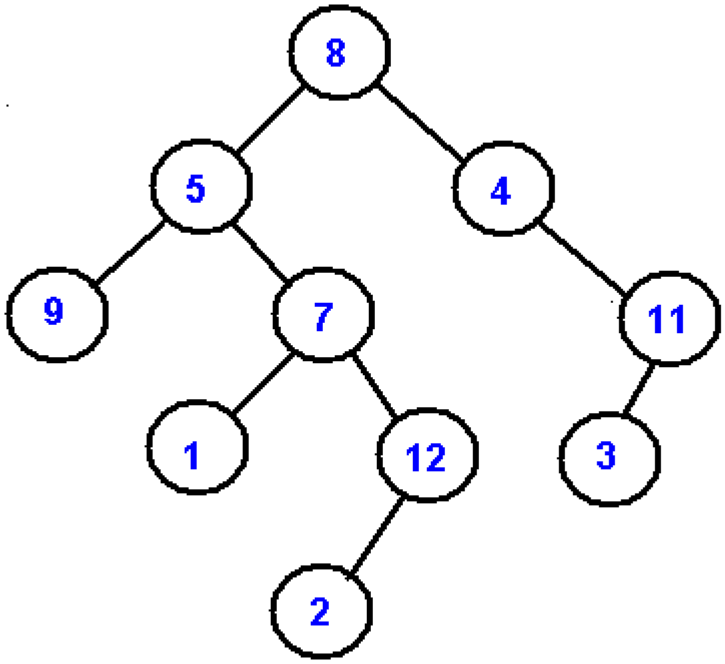
PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

InOrder traversal - visit the left child, then the parent and the right child;



- InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

PostOrder traversal - visit left child, then the right child and then the parent;



PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.

UNIT IV

❖ Graph Introduction

Graph is a dynamic data structure that is used in many applications like mathematics, geography, electrical engineering and computer science.

In fact, graph is a kind of tree with or without cycle. Basically graph theory was originated in Konigsberg Bridge problem by Leonhard Euler in the early 18th century.

❖ Graph Definition

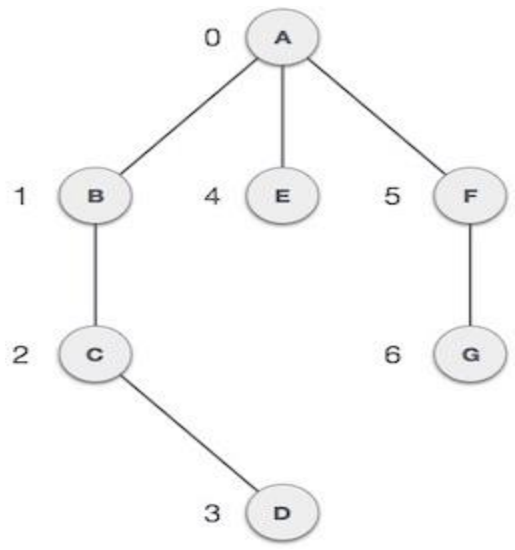
Graphs are a set of finite number of vertices and joining edges. A graph is a non – linear data structure as an element not follow single element, In a graph one element can follow multiple element and can be followed by multiple element. Graph data structure can be seen in real life models**. For Example**, road map of city. Some time it looks like a tree, as all tree come under the graph data structure but tree does not create cycle. Thus an acyclic graph is a tree.

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



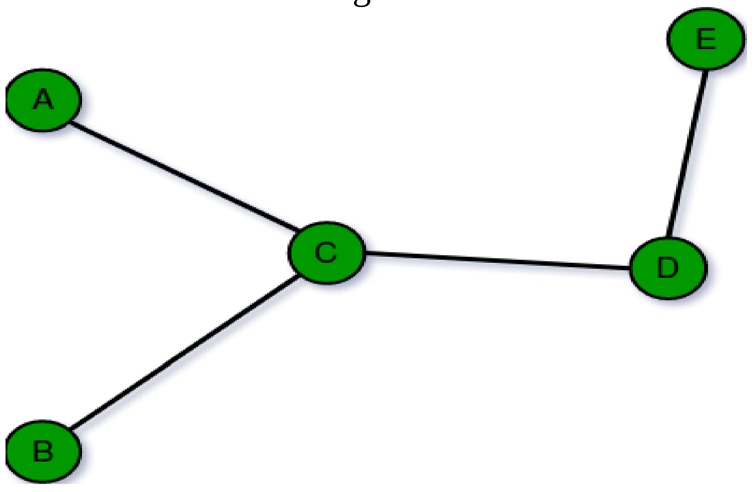
Basic Operations

Following are basic primary operations of a Graph −

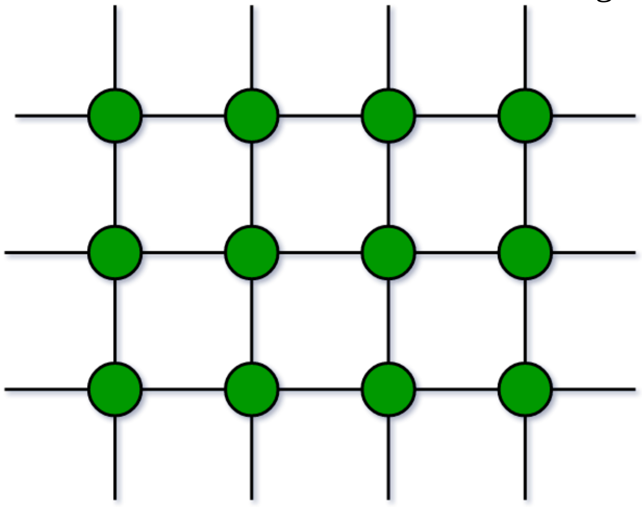
- **Add Vertex** − Adds a vertex to the graph.
- **Add Edge** − Adds an edge between the two vertices of the graph.
- **Display Vertex** − Displays a vertex of the graph

Types of Graph:

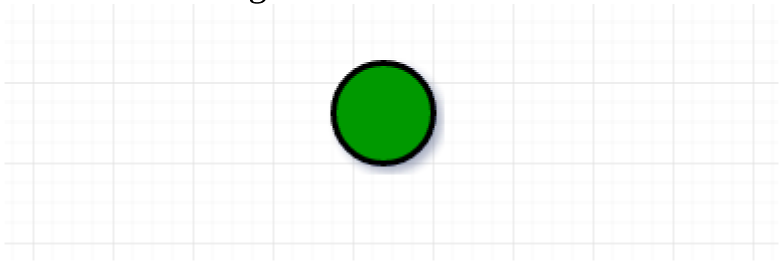
- **Finite Graphs:** A graph is said to be finite if it has finite number of vertices and finite number of edges.



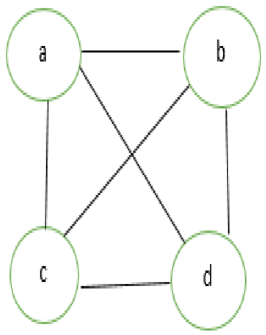
- **Infinite Graph:** A graph is said to be infinite if it has infinite number of vertices as well as infinite number of edges.



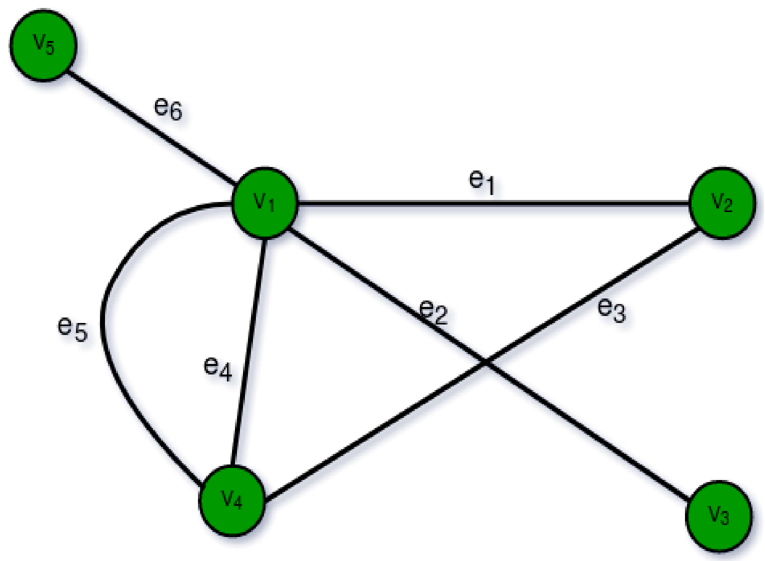
- **Trivial Graph:** A graph is said to be trivial if a finite graph contains only one vertex and no edge.



- **Simple Graph:** A simple graph is a graph which does not contains more than one edge between the pair of vertices. A simple railway tracks connecting different cities is an example of simple graph.

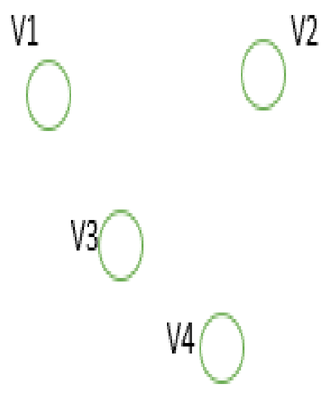


- **Multi Graph:** Any graph which contain some parallel edges but doesn't contain any self-loop is called multi graph. For example A Road Map.

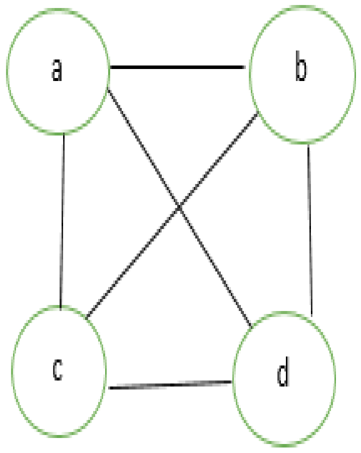


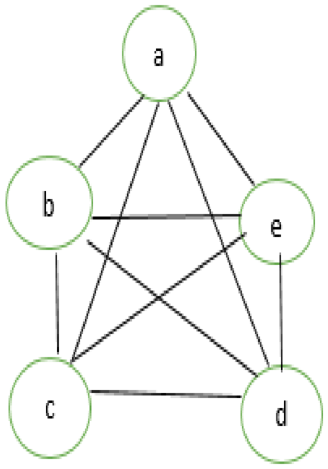
- **Parallel Edges:** If two vertices are connected with more than one edge than such edges are called parallel edges that is many roots but one destination.
- **Loop:** An edge of a graph which join a vertex to itself is called loop or a self-loop.

Null Graph: A graph of order n and size zero that is a graph which contain n number of vertices but do not contain any edge.

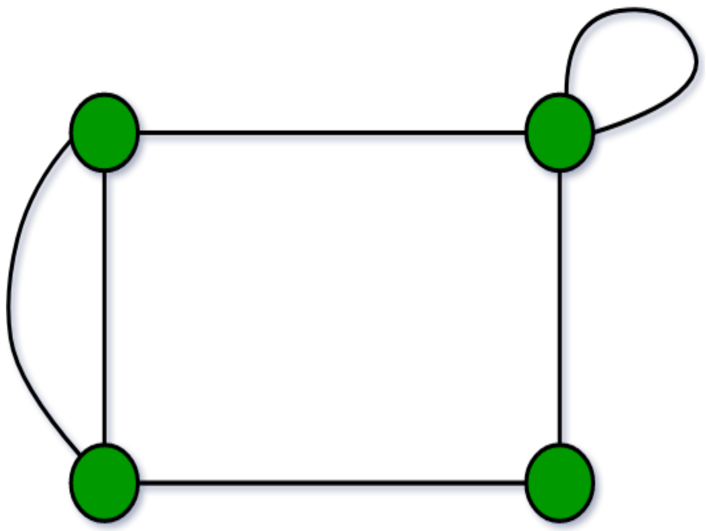


Complete Graph: A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attach with n-1 edges. A complete graph is also called Full Graph.

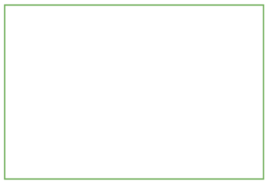


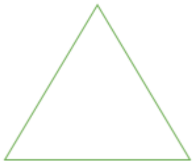


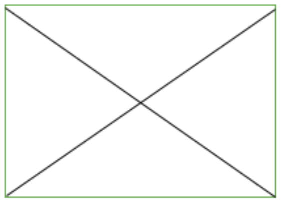
Pseudo Graph: A graph G with a self loop and some multiple edges is called pseudo graph.



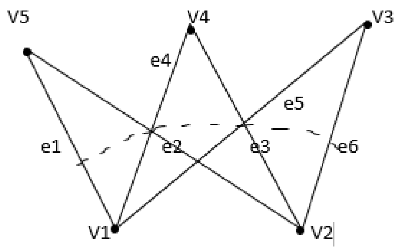
Regular Graph: A simple graph is said to be regular if all vertices of a graph G are of equal degree. All complete graphs are regular but vice versa is not possible.



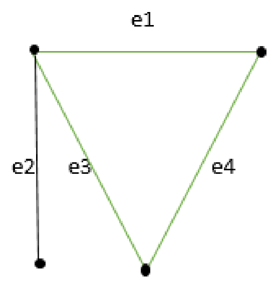




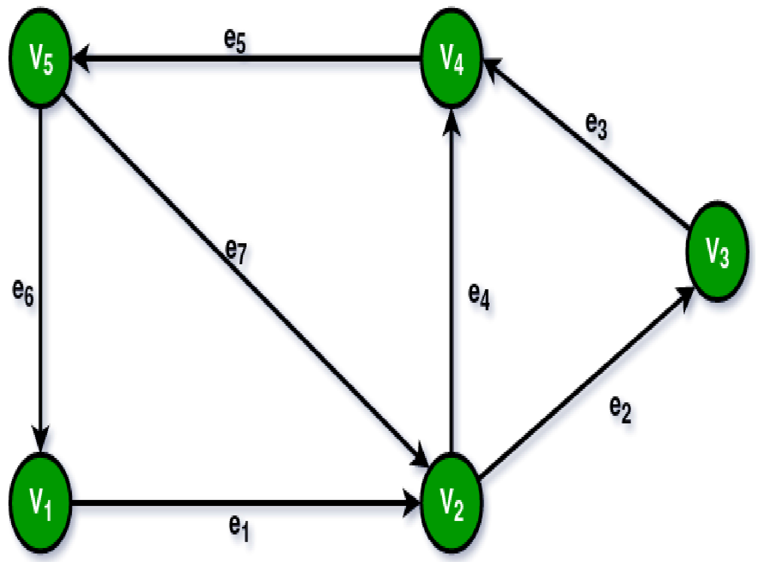
Bipartite Graph: A graph G = (V, E) is said to be bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way that each edge e of E(G) has its one end in V1(G) and other end in V2(G).
The partition V1 U V2 = V is called Bipartite of G.
Here in the figure:
V1(G)={V5, V4, V3}
V2(G)={V1, V2}



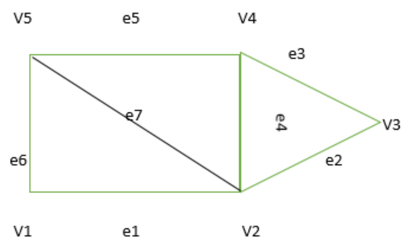
Labelled Graph: If the vertices and edges of a graph are labelled with name, data or weight then it is called labelled graph. It is also called *Weighted Graph*.



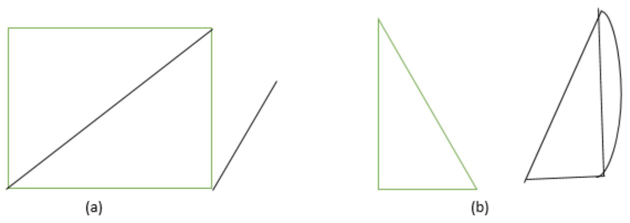
Digraph Graph: A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) is called Digraph. It is also called *Directed Graph*. Ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj.
Here in the figure:
e1 = (V1, V2)
e2 = (V2, V3)
e4 = (V2, V4)



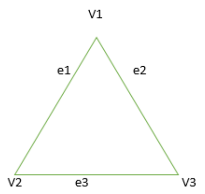
Subgraph: A graph G = (V1, E1) is called subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.



Connected or Disconnected Graph: A graph G is said to be connected if for any pair of vertices (Vi, Vj) of a graph G are reachable from one another. Or a graph is said to be connected if there exist atleast one path between each and every pair of vertices in graph G, otherwise it is disconnected. A null graph with n vertices is disconnected graph consisting of n components. Each component consist of one vertex and no edge.



Cyclic Graph: A graph G consisting of n vertices and n> = 3 that is V1, V2, V3- – – – – – – – – Vn and edges (V1, V2), (V2, V3), (V3, V4)- – – – – – – – – — -(Vn, V1) are called cyclic graph.

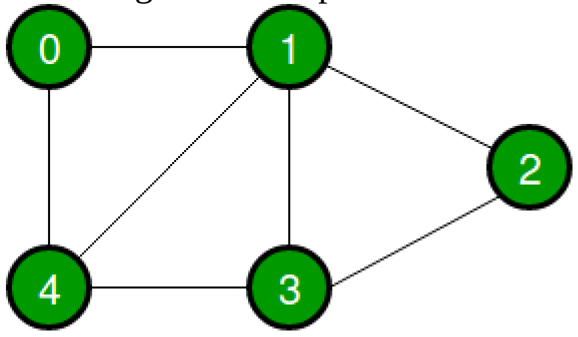


Application of Graphs:

- **Computer Science:** In computer science, graph is used to represent networks of communication, data organization, computational devices etc.
- **Physics and Chemistry:** Graph theory is also used to study molecules in chemistry and physics.
- **Social Science:** Graph theory is also widely used in sociology.
- **Mathematics:** In this, graphs are useful in geometry and certain parts of topology such as knot theory.
- **Biology:** Graph theory is useful in biology and conservation efforts.

❖ Representation to Graphs

A graph is a data structure that consists of the following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.
Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale. See this for more applications of graph.
Following is an example of an undirected graph with 5 vertices.



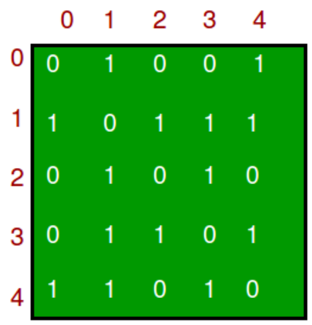
The following two are the most commonly used representations of a graph.
1. Adjacency Matrix
2. Adjacency List
There are other representations also like, Incidence Matrix and Incidence List. The

choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:



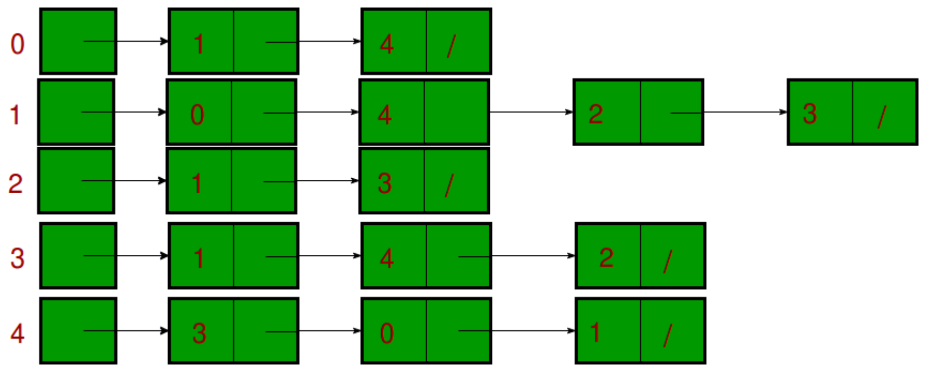
$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 & 1 & 0
\end{array}$$

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time. Please see this for a sample Python implementation of adjacency matrix.

Adjacency List:

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.



❖ Graph Traversals Shortest Path Algorithm

Shortest Path Routing

- In shortest path routing algorithm, a graph of the subnet is created.
- In this graph, each node represents the router and each are represents a link or communication line.
- In order to select the shortest path from a sender to receiver , the algorithm finds the shortest path between then on the graph.
- Several different metrics can be used to find out the shortest path between the router.
- One way of measuring path length is the number of hops i.e. this approach counts the number of intermediate routers that are lying in the path from sender to receiver.
- Other way is to find the total length of physical channel between a pair of routers.
- Various other metrics are also possible such as mean queuing and transmission delay. In this case, the path taking shortest time to deliver the packets from one router to the other may be chosen i.e. fastest path is the shortest path rather than the path with the fewest arcs or kilometers.
- The labels on the arcs can be computed as a function of the distance, bandwidth, average traffic, communication cost, mean queue length, measured delay etc.
- The algorithm weights various parameters and computer the shortest path based on any one or combination of criterions stated above.

❖ Dijkstra's Algorithm

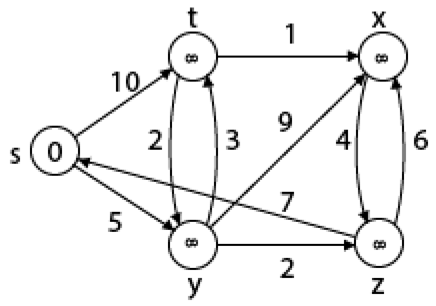
It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest - path estimate, inserts u into S and relaxes all edges leaving u.

Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the **greedy strategy**.

Analysis: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big - O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract - Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is O ($|V^2|+|E|=O(V^2)$).

Example:



Solution:

Step1: Q =[s, t, x, y, z]

We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices.

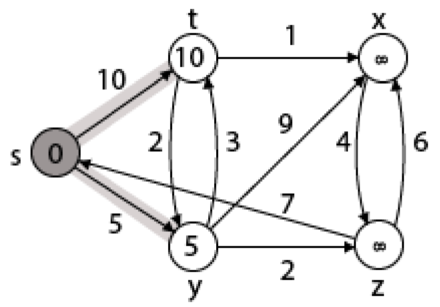
We make a stack, which contains those vertices which are selected after computation of shortest distance.

Firstly we take's' in stack M (which is a source)

1. M = [S]   Q = [t, x, y, z]

Step 2: Now find the adjacent of s that are t and y.

1. Adj [s] → t, y   [Here s is u and t and y are v]



Case - (i) s → t

d [v] > d [u] + w [u, v]
d [t] > d [s] + w [s, t]
∞ > 0 + 10   [false condition]

Then **d [t] ← 10**
π [t] ← 5

Adj [s] ← t, y

Case - (ii) s→ y

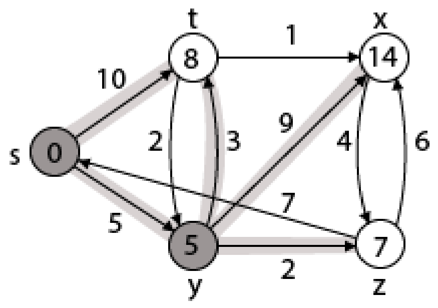
d [v] > d [u] + w [u, v]
d [y] > d [s] + w [s, y]
∞ > 0 + 5   [false condition]
∞ > 5

Then **d [y] ← 5**
π [y] ← 5

By comparing case (i) and case (ii)

Adj [s] → t = 10, y = 5
y is shortest

y is assigned in 5 = [s, y]



Step 3: Now find the adjacent of y that is t, x, z.

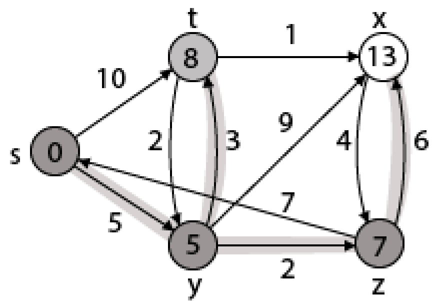
1. Adj [y] → t, x, z   [Here y is u and t, x, z are v]

Case - (i) y →t
d [v] > d [u] + w [u, v]
d [t] > d [y] + w [y, t]
10 > 5 + 3
10 > 8
Then d [t] ← 8
π [t] ← y

Case - (ii) y → x
d [v] > d [u] + w [u, v]
d [x] > d [y] + w [y, x]
∞ > 5 + 9
∞ > 14
Then d [x] ← 14
π [x] ← 14

Case - (iii) y → z
d [v] > d [u] + w [u, v]
d [z] > d [y] + w [y, z]
∞ > 5 + 2
∞ > 7
Then d [z] ← 7
π [z] ← y

By comparing case (i), case (ii) and case (iii)
Adj [y] → x = 14, t = 8, z =7
z is shortest **z is assigned in 7 = [s, z]**

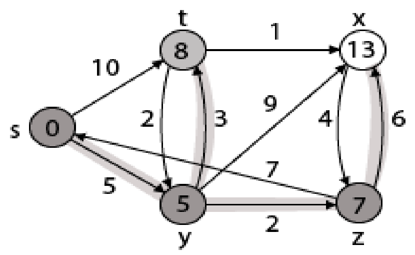


Step - 4 Now we will find adj [z] that are s, x

1. Adj [z] → [x, s]   [Here z is u and s and x are v]

Case - (i) z → x
d [v] > d [u] + w [u, v]
d [x] > d [z] + w [z, x]
14 > 7 + 6
14 > 13
Then   d [x] ← 13
π [x] ← z

Case - (ii) z → s
d [v] > d [u] + w [u, v]
d [s] > d [z] + w [z, s]
0 > 7 + 7
0 > 14
∴ This condition does not satisfy so it will be discarded.
Now we have x = 13.



Step 5: Now we will find Adj [t]

Adj [t] → [x, y] [Here t is u and x and y are v]

Case - (i) t → x

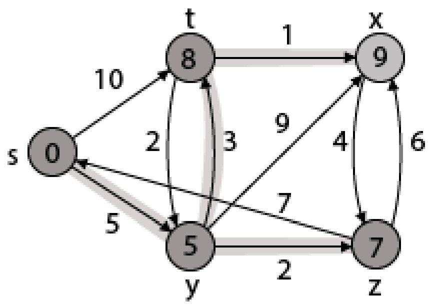
d [v] > d [u] + w [u, v]
d [x] > d [t] + w [t, x]
13 > 8 + 1
13 > 9

Then **d [x] ← 9**
π [x] ← t

Case - (ii) t → y

d [v] > d [u] + w [u, v]
d [y] > d [t] + w [t, y]
5 > 10

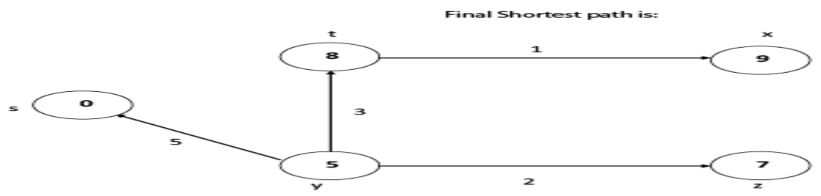
∴ This condition does not satisfy so it will be discarded.



Thus we get all shortest path vertex as

Weight from s to y is 5
Weight from s to z is 7
Weight from s to t is 8
Weight from s to x is 9

These are the shortest distance from the source's' in the given graph.



Disadvantage of Dijkstra's Algorithm:

1. It does a blind search, so wastes a lot of time while processing.

2. It can't handle negative edges.
3. It leads to the acyclic graph and most often cannot obtain the right shortest path.
4. We need to keep track of vertices that have been visited

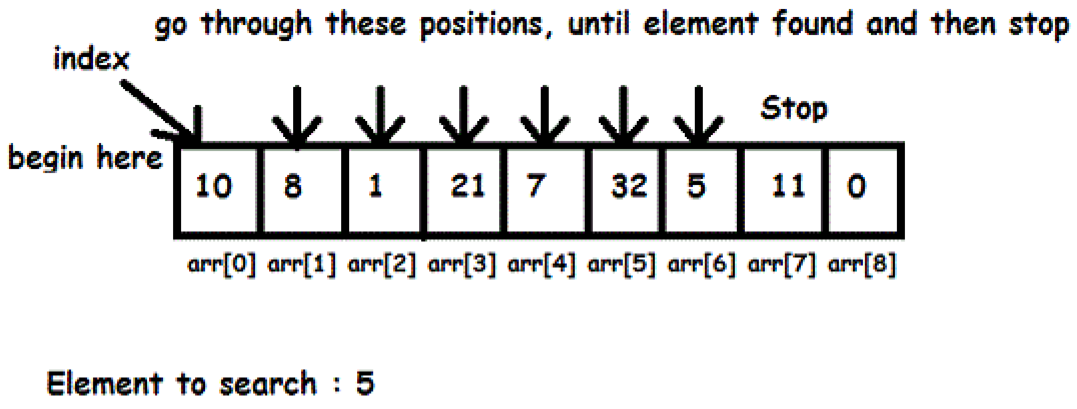
❖ Searching

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

1. Linear Search or Sequential Search
2. Binary Search

What is Linear Search?

This is the simplest method for searching. In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached. As against this, searching in case of unsorted list also begins from the 0th element and continues until the element or the end of the list is reached.

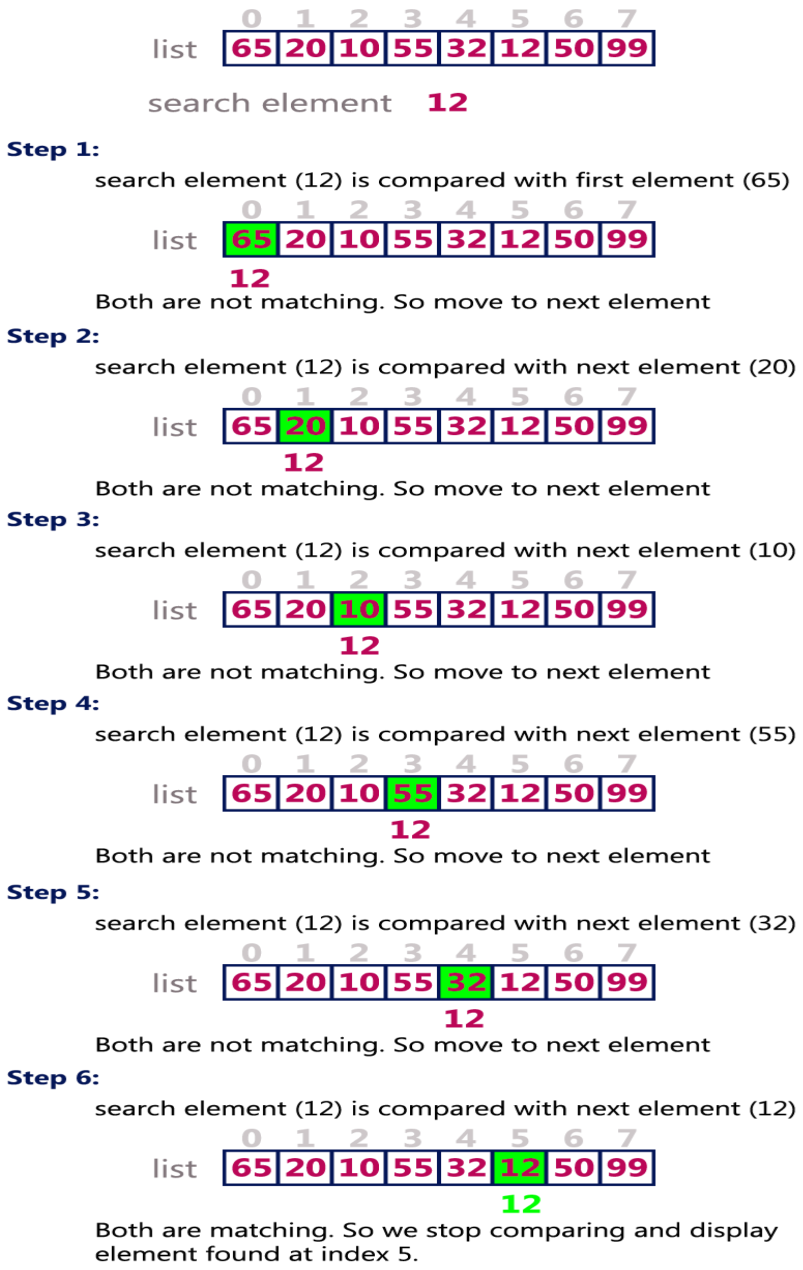


Linear search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.

- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function. **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Let see Example of Linear Search


0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
search element 12

Step 1:
search element (12) is compared with first element (65)
list 65 20 10 55 32 12 50 99
12
Both are not matching. So move to next element

Step 2:
search element (12) is compared with next element (20)
list 65 20 10 55 32 12 50 99
12
Both are not matching. So move to next element

Step 3:
search element (12) is compared with next element (10)
list 65 20 10 55 32 12 50 99
12
Both are not matching. So move to next element

Step 4:
search element (12) is compared with next element (55)
list 65 20 10 55 32 12 50 99
12
Both are not matching. So move to next element

Step 5:
search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:
search element (12) is compared with next element (12)

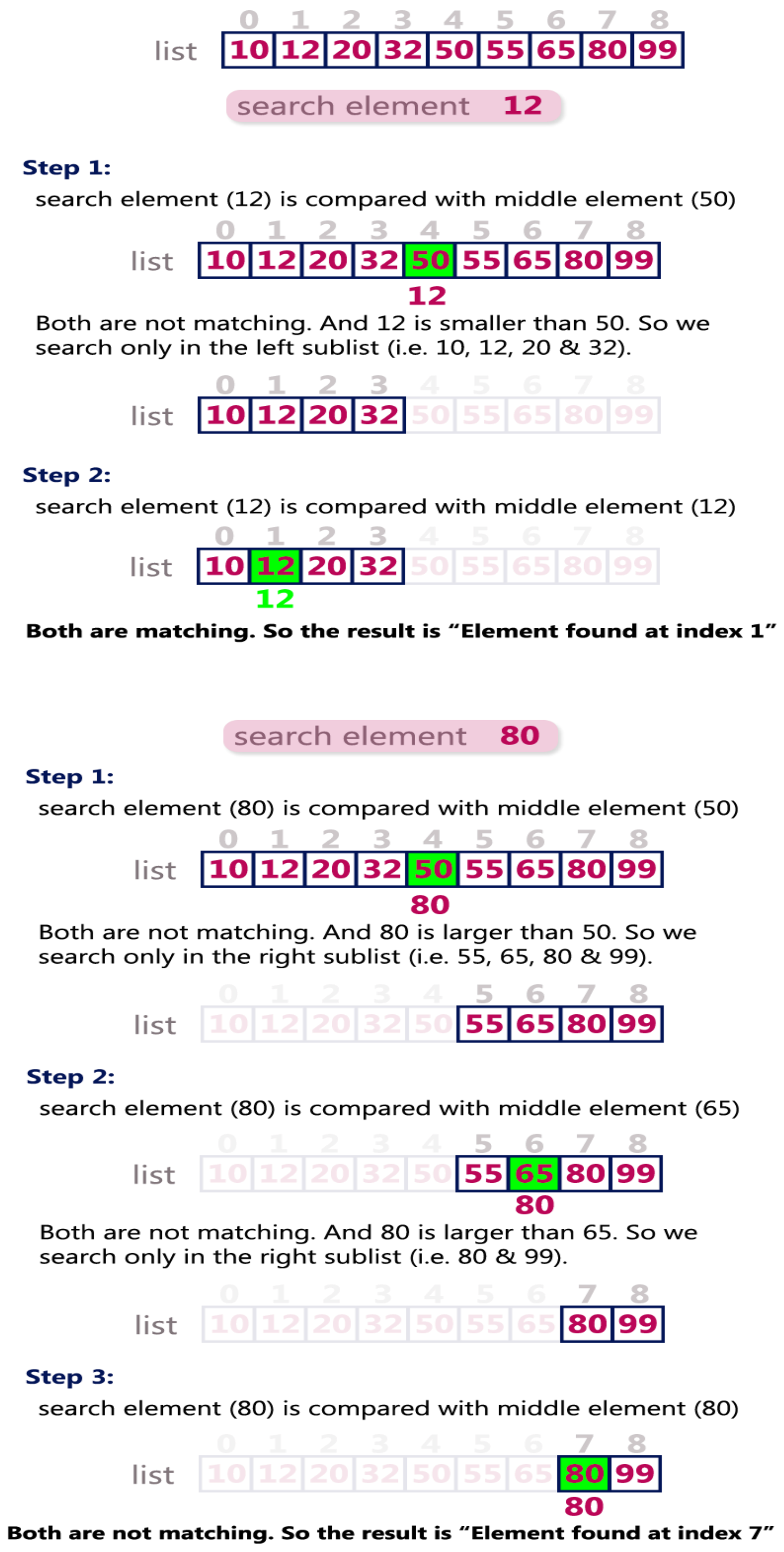
Both are matching. So we stop comparing and display element found at index 5.

Binary Search Algorithm

Binary search algorithm finds a given element in a list of elements with **O(log n)**time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only a sorted list of elements. That means the binary search is used only with a list of elements that are already arranged in an order. The binary search can not be used for a list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for the left sublist of the middle element. If the search element is larger, then we repeat the same process for the right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Find the middle element in the sorted list.
- **Step 3 -** Compare the search element with the middle element in the sorted list. **Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.
- **Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.
- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9 -** If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.
  Example
- Consider the following list of elements and the element to be searched.


0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99
search element 12

Step 1:

search element (12) is compared with middle element (50)

list 10 12 20 32 50 55 65 80 99
12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

Step 1:

search element (80) is compared with middle element (50)

list 10 12 20 32 50 55 65 80 99
80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

❖ Sorting

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Techniques of Sorting

1. Insertion Sort Algorithm
2. Selection Sort Algorithm
3. Bubble sort

Insertion Sort Algorithm

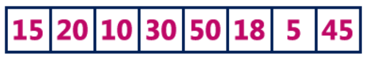
Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

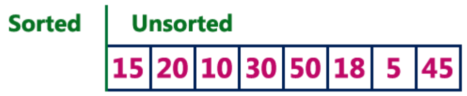
The insertion sort algorithm is performed using the following steps...

- Step 1 - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- Step 2: Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- Step 3: Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.
- **Complexity of the Insertion Sort Algorithm**:To sort an unsorted list with **'n'** number of elements, we need to make **(1+2+3+......+n-1) = (n (n-1))/2** number of comparisons' in the worst case. If the list is already sorted then it requires **'n'** number of comparisons'.
- **Worst Case : O(n^2)**
  Best Case : Ω(n)
  Average Case : Θ(n^2)
- **Example of Insertion Sort**

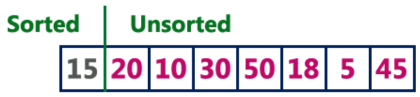
Consider the following unsorted list of elements...



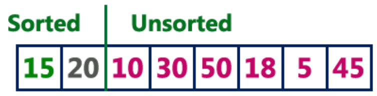
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...



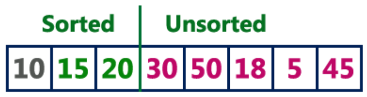
Move the first element 15 from unsorted portion to sorted portion of the list.



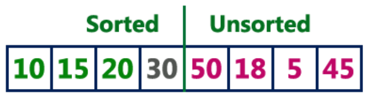
To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position



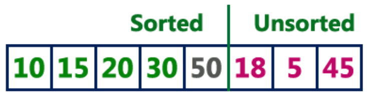
To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



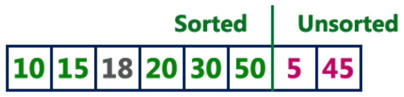
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



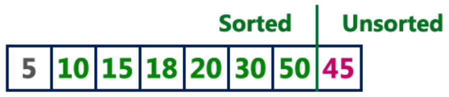
To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



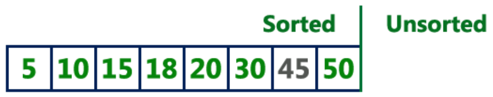
To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



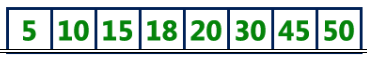
To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.



To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...



Selection Sort Algorithm

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order. Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using the following steps...

- **Step 1 -** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

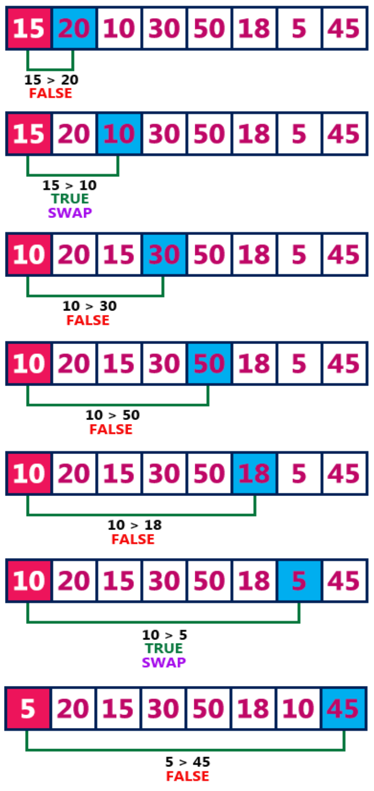
  Complexity of the Selection Sort Algorithm
- To sort an unsorted list with **'n'** number of elements, we need to make **((n-1)+(n-2)+(n-3)+......+1) = (n (n-1))/2** number of comparisions in the worst case. If the list is already sorted then it requires **'n'** number of comparisions.
- **Worst Case : O(n^2)**
  Best Case : Ω(n^2)
  Average Case : Θ(n^2)

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.



List after 1st iteration

5	20	15	30	50	18	10	45

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

5	10	20	30	50	18	15	45

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

5	10	15	30	50	20	18	45

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

5	10	15	18	50	30	20	45

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

5	10	15	18	20	50	30	45

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

5	10	15	18	20	30	50	45

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration

5	10	15	18	20	30	45	50

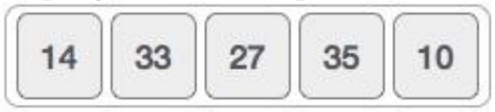
Final sorted list

Bubble sort

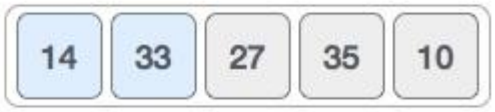
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2) where **n** is the number of items.

How Bubble Sort Works?

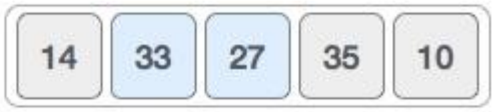
We take an unsorted array for our example. Bubble sort takes O(n^2) time so we're keeping it short and precise.



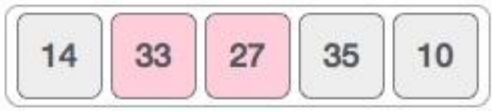
Bubble sort starts with very first two elements, comparing them to check which one is greater.



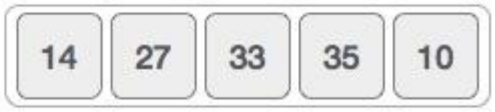
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



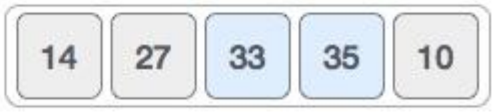
We find that 27 is smaller than 33 and these two values must be swapped.



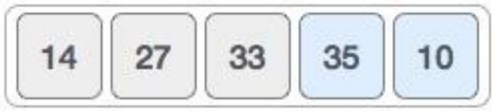
The new array should look like this −



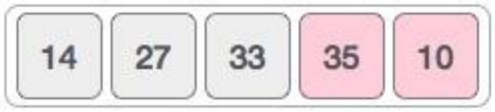
Next we compare 33 and 35. We find that both are in already sorted positions.



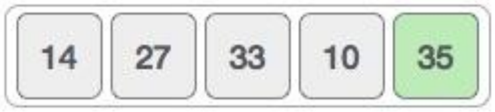
Then we move to the next two values, 35 and 10.



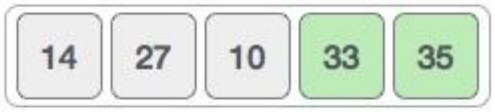
We know then that 10 is smaller 35. Hence they are not sorted.



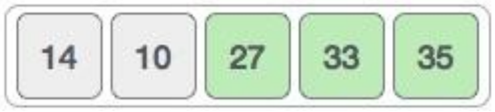
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



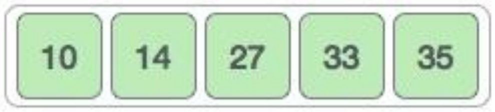
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

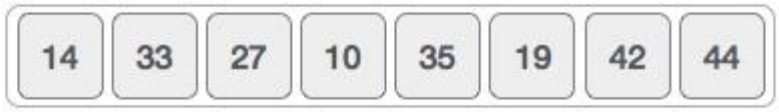
Merge sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

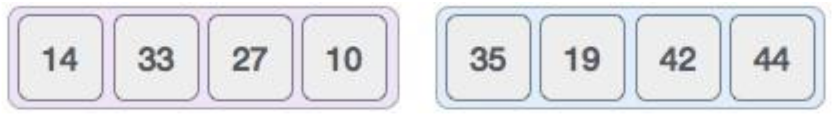
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

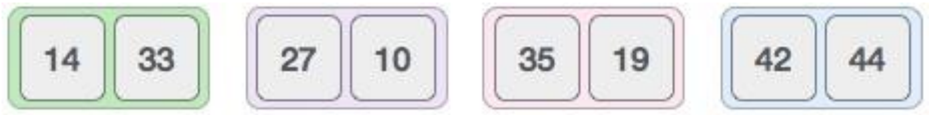
To understand merge sort, we take an unsorted array as the following −



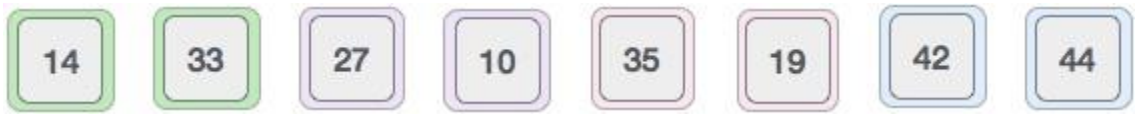
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

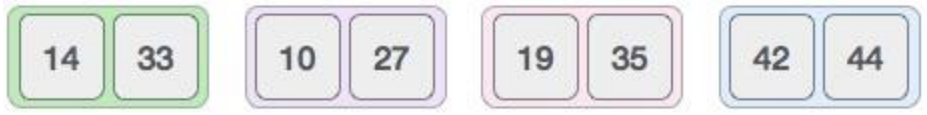


We further divide these arrays and we achieve atomic value which can no more be divided.

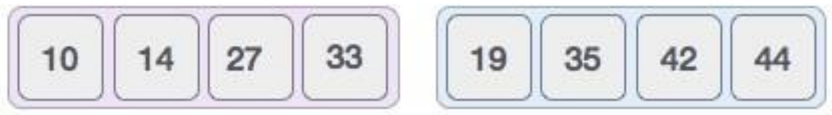


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

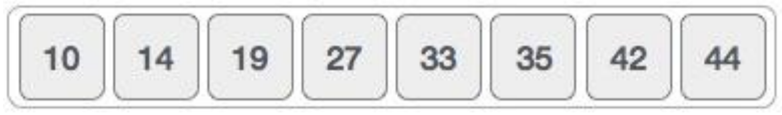
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −



Now we should learn some programming aspects of merge sorting.

QuickSort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

Always pick first element as pivot.

Always pick last element as pivot (implemented below)

Pick a random element as pivot.

Pick median as pivot.

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

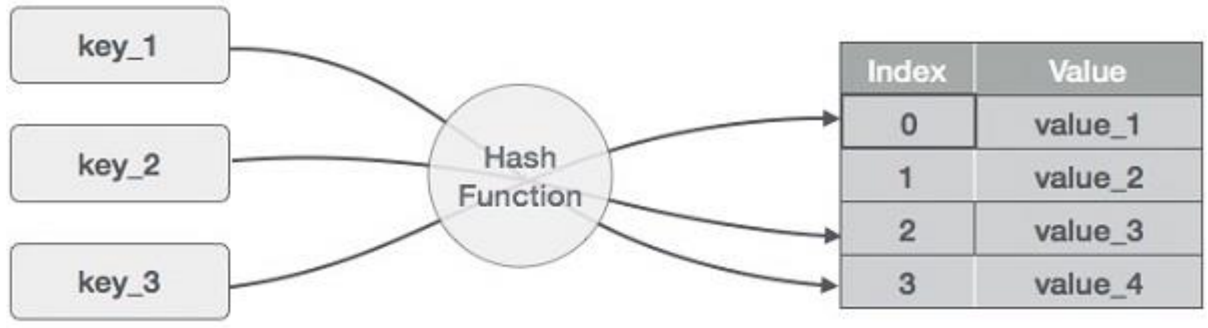
❖ Hash Table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

❖ Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an **example** of hash table of size 20, and the following items are to be stored. Item are in the (key, value) format.



(1,20)

(2,70)

(42,80)

(4,25)

(12,44)

(14,32)

(17,11)

(13,78)

(37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12

6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13

9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

Search − Searches an element in a hash table.

Insert − inserts an element in a hash table.

delete − Deletes an element from a hash table.

Advantage-

Unlike other searching techniques,

Hashing is extremely efficient.

The time taken by it to perform the search does not depend upon the total number of elements.

It completes the search with constant time complexity O(1).

Hashing Mechanism-

In hashing,

An array data structure called as **Hash table** is used to store the data items.

Based on the hash key value, data items are inserted into the hash table.

❖ Types of Hash Functions

There are various types of hash functions available such as-

Mid Square Hash Function

Division Hash Function

Folding Hash Function etc

1. Division method

In this the hash function is dependent upon the remainder of a division. **For example:-**if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.

Then:

h(key)=record% table size.

2=52%10

8=68%10

9=99%10

4=84%10

2. Mid square method

In this method firstly key is squared and then mid part of the result is taken as the index**. For example**: consider that if we want to place a record of 3101 and the size of table is 1000. So 3101*3101=9616201 i.e. **h (3101) = 162 (middle 3 digit)**

3. Digit folding method

In this method the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key. **For example:** consider a record of 12465512 then it will be divided into parts i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

H(key)=124+655+12

=791

Characteristics of good hashing function

The hash function should generate different hash values for the similar string.

The hash function is easy to understand and simple to compute.

The hash function should produce the keys which will get distributed, uniformly over an array.

A number of collisions should be less while placing the data in the hash table.

The hash function is a perfect hash function when it uses all the input data.

❖ Collision

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

Collision resolution technique

If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below.

1) Chaining

It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

Example: Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are 31,33,77,61. In the above diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

2) Linear probing

It is very easy and simple method to resolve or to handle the collision. In this collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table size. Consider that following keys are to be inserted that are 56,64,36,71.

In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

3) Quadratic probing

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the H(key)=(H(key)+x*x)%table size. Let us consider we have to insert following elements that are:-67, 90,55,17,49.

In this we can see if we insert 67, 90, and 55 it can be inserted easily but at case of 17 hash function is used in such a manner that :-(17+0*0)%10=17 (when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so (17+1*1)%10=8.in this case bucket 8 is empty hence we will place 17 at index 8.

4) Double hashing

It is a technique in which two hash function are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

It must never evaluate to zero.

Must sure about the buckets, that they are probed.

The hash functions for this technique are:

H1(key)=key % table size

H2(key)=P-(key mod P)

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

Example: Let us consider we have to insert 67, 90,55,17,49.

In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 again the bucket is full and in this case we have to use the second hash function which is H2(key)=P-(key mode P) here p is a prime number which should be taken smaller than the hash table so value of p will be the 7.

i.e. H2(17)=7-(17%7)=7-3=4 that means we have to take 4 jumps for placing the 17. Therefore 17 will be placed at index 1.

❖ Perfect Hashing

Definition of Perfect Hashing

Perfect hashing is defined as a model of hashing in which any set of n elements can be stored in a hash table of equal size and can have lookups performed in constant time. It was specifically invented and discussed by Fredman, Komlos and Szemeredi (1984) and has therefore been nicknamed as "FKS Hashing".

Definition of Static Hashing

Static Hashing defines another form of the hashing problem which permits users to accomplish lookups on a finalized dictionary set (that means all objects in the dictionary are final as well as not changing).

Application

Since static hashing needs that the database, its objects and reference remain the same its applications are limited. Databases which contain information which experiences rare change are also eligible as it would only require a full rehash of the whole database on rare occasion. Various examples of this hashing scheme include sets of words and definitions of specific languages, sets of significant data for an organization's personnel, etc.