ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATIONS) ACT, 2016

RAJAMPET, Annamayya District, AP, INDIA

Course : Data Structures and Analysis of Algorithms

Course Code : **24FMCA11T**

Branch : MCA

Prepared by : **S. Mastan**

Designation : **Assistant Professor**

Department : **MCA**

ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATIONS) ACT, 2016

RAJAMPET, Annamayya District, AP, INDIA

Title of the Course : Data Structures and Analysis of Algorithms
Category : PC
Year : I
Semester : I
Course Code : 24FMCA11T
Branch : MCA

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	-	-	3

Course Objectives:

- To design and implement Linear data structure.
- To design and implement Non-Linear data structures.
- To understand the graph algorithms.
- To analyze the efficiency of algorithms and using Dynamic Programming technique to solve the problems.
- To solve the problems using Backtracking Technique and understanding the NP-Hard problems and NP-complete problems.

Course Outcomes:

The Student will be able to

1. Summarize the Linear data structures.
2. Comprehend the Non-linear data structures.
3. Summarize the Graph concepts.
4. Apply dynamic programming technique to solve the problems.
5. Apply backtracking technique for solving the problems.

Unit 1 LINEAR DATA STRUCTURES **10 Hrs**

Introduction- Definition, ARRAYS, LINKED LISTS, STACKS AND QUEUES: Arrays: Implementation, Operations, Applications. Linked List: Implementation, Operations, Applications. Stack: Implementation, Operations, Applications. Queues: Implementation, Operations, Applications.

Unit 2 NON-LINEAR DATA STRUCTURE AND HASHIN **12 Hrs**

Introduction- Definition and Basic terminologies of trees and binary trees, Representation of trees, Binary Tree traversals, Binary Search Trees: Definition, Operations and applications. Height Balanced Trees or AVL. Threaded binary trees. Hashing: The symbol table, Hashing Functions, Collision Resolution Techniques.

Unit 3 Graphs **10 Hrs**

Introduction - Representation of graph - Graph Traversals: Depth- First Search and Breadth-First Search, Applications of graphs, Shortest-path algorithms: Dijkstra's Algorithm, Floyd Warshal's Algorithm.Spanning Trees , Minimum spanning trees: Prim's Algorithm and Kruskal's Algorithm.

Unit 4 ALGORITHM ANALYSIS & DYNAMIC PROGRAMMING 12 Hrs

Algorithm Analysis – Asymptotic Notations. Divide and Conquer: General Method, Merge Sort, Quick Sort, Binary Search. Dynamic Programming: Multistate Graphs, Optimal Binary Search Tree, 0/1Knapsack Problem, Travelling Salesman Problem.

Unit 5 BACKTRACKING AND NP-COMPLETENESS 12 Hrs

Backtracking: General Method, 8-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles. Assignment Problem: P & NP problem, NP-Hard, NP-complete problems, Reduction, Satisfiability Problem, Approximation algorithms for NP-hard problem, COOK'S Theorem.

PRESCRIBED TEXTBOOKS:

1. Data Structures and Algorithms Analysis in C, Mark Allen Weiss, 2nd, Edition ,2023,
2. Ellis Horowitz, SartajSahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Universities Press , 2nd Edition, 2019.

REFERENCE BOOKS:

1. A.M.Tenenbaum, Y. Langsam, M.J.Augenstein -Data Structures using C, Pearson, 1st Edition , 2003
2. Reema Thareja, Data Structures Using C , Oxford, 2nd Edition, 2014.
3. Yashavant Kanetkar, Data Structures through C, BPB Publications, 4th Edition, 2022.

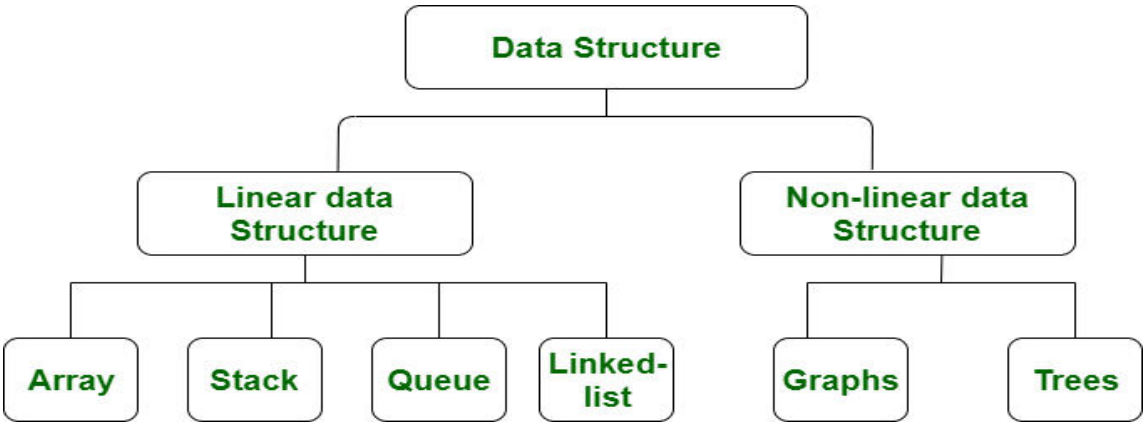
CO-PO MAPPING:

Course Outcomes	Foundation Knowledge	Problem Analysis	Development of Solutions	Modern Tool Usage	Individual and Teamwork	Project Management and Finance	Ethics	Life-long Learning
24FMCA11T.1	2	2	1	1	-	-	-	-
24FMCA11T.2	2	2	1	1	-	-	-	-
24FMCA11T.3	2	2	1	1	-	-	-	-
24FMCA11T.4	3	2	1	2	-	-	-	-
24FMCA11T.5	3	2	1	2	-	-	-	-

UNIT - I

What is the Linear data structure?

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.



The types of linear data structures are Array, Queue, Stack, Linked List.

- **Array:** An array consists of data elements of a same data type. For example, if we want to store the roll numbers of 10 students, so instead of creating 10 integer type variables, we will create an array having size 10. Therefore, we can say that an array saves a lot of memory and reduces the length of the code.
- **Stack:** It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.
- **Queue:** It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue **front** end and **rear** The insertion operation performed at the back end is known ad enqueue, and the deletion operation performed at the front end is known as dequeue.
- **Linked list:** It is a collection of nodes that are made up of two parts, i.e., data element and reference to the next node in the sequence.

Difference between Linear and Non-linear Data Structures:

Linear Data Structure	Non-linear Data Structure
In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

Non-Linear Data Structure:

- When the data elements are organised in some arbitrary function without any sequence, such data structures are called non-linear data structures.
- Examples of such type are trees, graphs.
- The relationship of adjacency is not maintained between elements of a non-linear data structure.

What are Arrays?

An array is a data structure for storing more than one data item that has a similar data type. The items of an array are allocated at adjacent memory locations. These memory locations are called **elements** of that array.

The total number of elements in an array is called **length**. The details of an array are accessed about its position. This reference is called **index** or **subscript**.

(or)

An *array in data structure* is static data structure used to store data of homogeneous (same) type.

- ✓ Homogeneous simply says that an array can be used to store only integer values or only float values or only characters, or String, etc.
- ✓ Static nature means array size is defined at the beginning and it cannot grow or shrink at later stage.

Definition

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject. instead of that, we can define an array which can store the marks in each subject at a the contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in first subject, **marks[1]** denotes the marks in 2nd subject and so on.

Following are the important terms to understand the concept of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

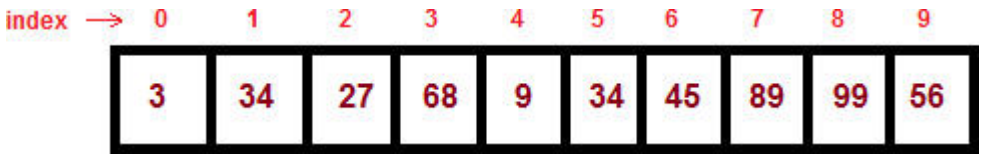
Basic concept of Array

Listed below are important and *basic concept of array*:

- ✓ Array is static in nature. Its size is defined at the beginning and cannot increase or decrease based on the requirement.
- ✓ All the elements in array are stored in **consecutive** or **contiguous memory location**.
- ✓ Data type of an array is declared only at the beginning during initialization. So an array can either be declared as int, float, char, etc.
- ✓ An array can hold multiple values of same type under single reference name, which is name of the array.
- ✓ An array is **index based** data structure.
- ✓ Each element in array is stored at a particular index and could be accessed using the index easily.

- Array can be used to create complex data structures such as stack and queues.

Given below is example and diagrammatic representation of array.



Points to note from above diagram about array:

1. Array index starts from 0
2. First element 3 is stored at index 0, second element 34 at index 1 and so on.
3. Above array is of length 10 and thus is capable of storing maximum 10 element.
4. We can retreive elements from any position using index from the above array.
5. We can delete, search and insert the element in array.

Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

for example, in C language, the syntax of declaring an array is like following:

int arr[10]; **char** arr[10]; **float** arr[5]

Program without array:

```
#include <stdio.h>
void main ()
{
   int marks_1 = 56, marks_2 = 78, marks_3 = 88, marks_4 = 76, marks_ = 56, marks
_6 = 89;
float avg = (marks_1 + marks_2 + marks_3 + marks_4 + marks_5 +marks_6) / 6 ;
   printf(avg);
}
```

Program by using array:

```
#include <stdio.h>
void main ()
```

```
{
   int marks[6] = {56,78,88,76,56,89);
   int i;
   float avg;
   for (i=0; i<6; i++ )
   {
      avg = avg + marks[i];
   }
   printf(avg);
}
```

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.

Some more Advantages of array

Advantages of array data structure are:

1. all elements are stored strategically based on index number in well organized manner.
2. we can easily traverse (move back and forth or sideways) the above array
3. we can fetch element based on their index number at any time.
4. we can also search for a particular element from array by traversing it.
5. we can sort the above array in ascending or descending order using various sorting techniques.
6. we can delete any element from the above array.

Disadvantages of array

Disadvantages of array data structure are:

1. Arrays static nature can be a blocker in development because in real life projects need for array size may increase or decrease.

2. If array size requirement increases while development is in progress then we need to discard the array on which work is in progress and declare new array that meets the size requirement. But this may later again change as application grows.

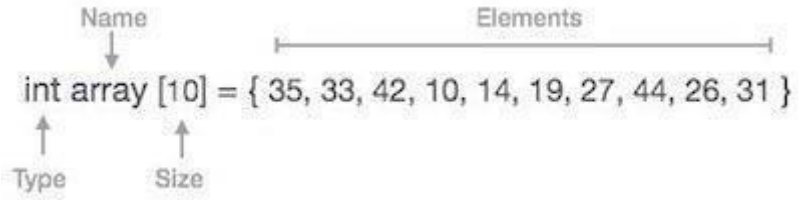
3. If the size requirement is less then memory wastage is on the card. For example, you declared array of size 100 but later found out that you only needed to store 82 elements so its a complete wastage of 18 elements storage space. Because when you declared array with size 100, memory space to store 100 element was already blocked

on disk.

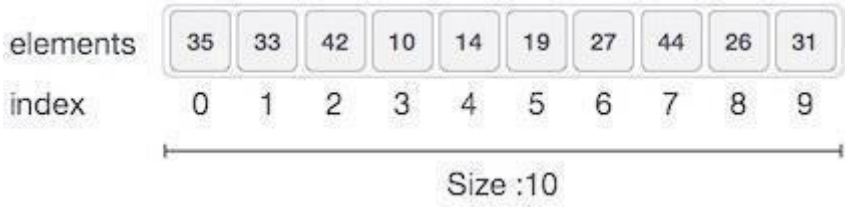
4. Array is homogenous in nature which means element of only one type can be stored in one array. But in real life scenarios, we may be required to store elements of different types. For example, details about student such as name (String), age(integer), subject(String), etc. But array doesn't have such facility.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Important concept of array:

- Array always fixed size of data elements (Declaration)
- Array can be considered as a static data
- Array elements are placed in sequential order thus eliminating the gap in the memory allocation
- The index value starts with 0
- The data elements should be of same data type
- Array can be used in search algorithms (Binary search)
- Array can be used in sorting algorithms (Bubble sort)
- An array can be declared, initialized and referred by value.

Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

```
int marks[5]={20,30,40,50,60};
```

```
#include<stdio.h>
int main(){
int i=0;
int marks[5];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]
=70;
marks[3]
=85;
marks[4]
=75;
//traversal of array
for(i=0;i<5;i++){
printf("%d
\n",marks[i]);
}//end of for loop
return 0;
}
```

User Defined Elements through Keyborad:

```
#include<stdio.h>

int main()
{
   int arr[5];              //array of lengeth 5 type integer.. arr[0]-->arr[4]
   int i;

   printf("Enter 5 integers\n");
   for(i = 0; i < 5; i++)      //for getting 5 elements from user
   scanf("%d",&arr[i]);

   printf("The elements are\n");
   for(i = 0; i < 5; i++)     //printing all 5 elements
   printf("%d\n",arr[i]);

   return 0;
}
```

Types of Arrays

The various types of arrays are as follows.

- One dimensional array
- Multi-dimensional array

One-Dimensional Array
A one-dimensional array is also called a single dimensional array where the elements will be accessed in sequential order. This type of array will be accessed by the subscript of either a column or row index.

Multi-Dimensional Array

When the number of dimensions specified is more than one, then it is called as a multi-dimensional array. Multidimensional arrays include 2D arrays and 3D arrays.

2D Array

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

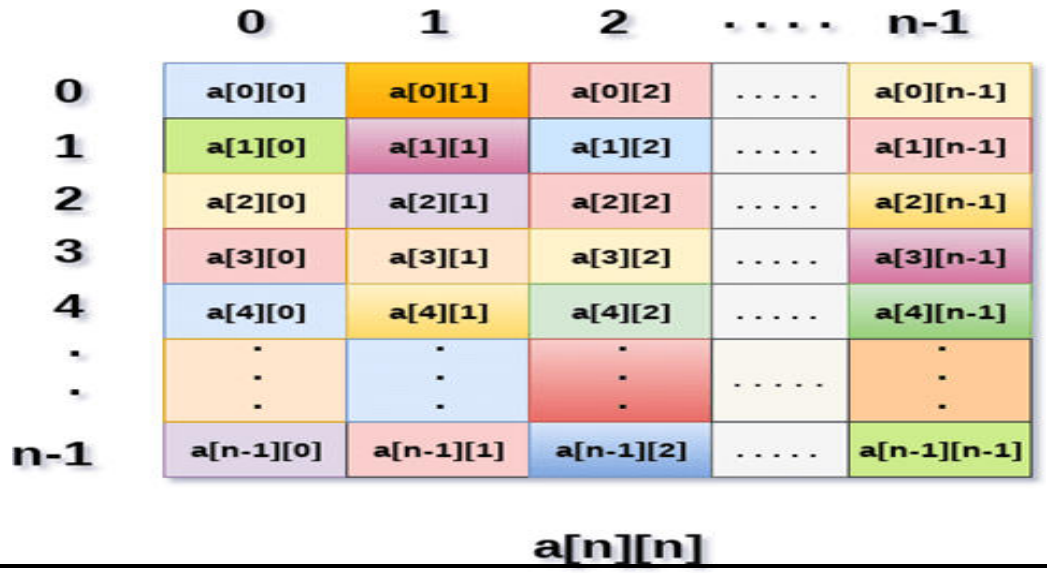
However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

How to declare 2D Array

The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.

1. **int** arr[max_rows][max_columns];

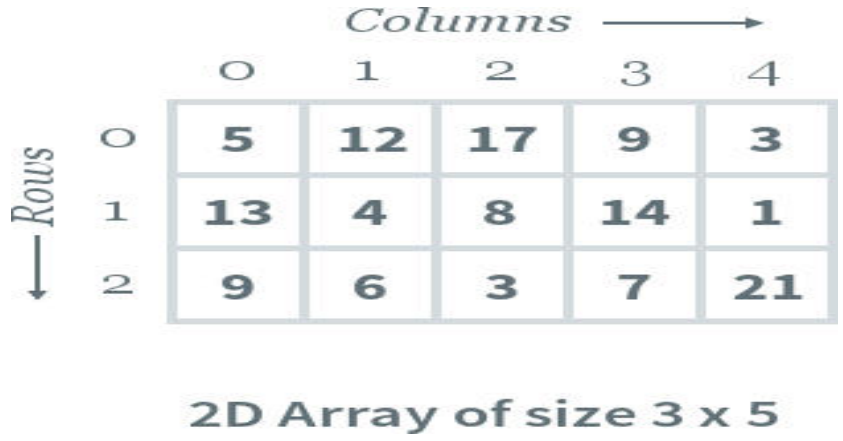
however, It produces the data structure which looks like following.



Multi-dimensional

A multi-dimensional **array** is an array of arrays. 2-dimensional arrays are the most commonly used. They are used to store data in a tabular manner.

Consider following 2D array, which is of the size 3×5. For an array of size N×M, the rows and columns are numbered from 0 to N−1 and columns are numbered from 0 to M−1, respectively. Any element of the array can be accessed by arr[i][j] where 0≤i<N and 0≤j<M. For example, in the following array, the value stored at arr[1][3] is 14.



2D Array of size 3 x 5

2D array declaration:

To declare a 2D array, you must specify the following:
Row-size: Defines the number of rows
Column-size: Defines the number of columns
Type of array: Defines the type of elements to be stored in the array i.e. either a number, character, or other such datatype. A sample form of declaration is as follows:

```
type arr[row_size][column_size]
```

A sample C array is declared as follows:

```
int arr[3][5];
```

2D array initialization:
An array can either be initialized during or after declaration. The format of initializing an array during declaration is as follows:

```
type arr[row_size][column_size] = {{elements}, {elements} ... }
```

An example in C is given below:

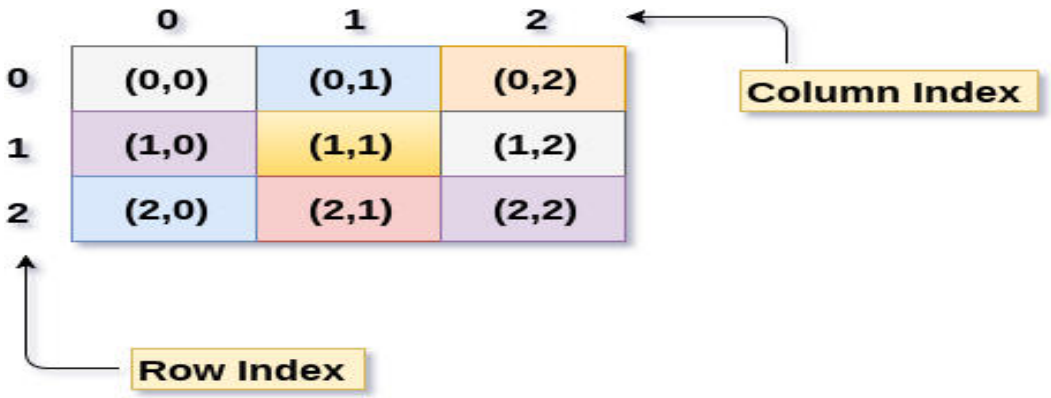
```
int arr[3][5] = {{5, 12, 17, 9, 3}, {13, 4, 8, 14, 1}, {9, 6, 3, 7, 21}};
```

Representation of ARRAYS:

However, 2 D arrays exists from the user point of view. 2D arrays are created to implement a relational database table lookalike data structure, in computer memory, the storage technique for 2D array is similar to that of an one dimensional array.

The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array. We do need to map two dimensional array to the one dimensional array in order to store them in the memory.

A 3 X 3 two dimensional array is shown in the following image. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.

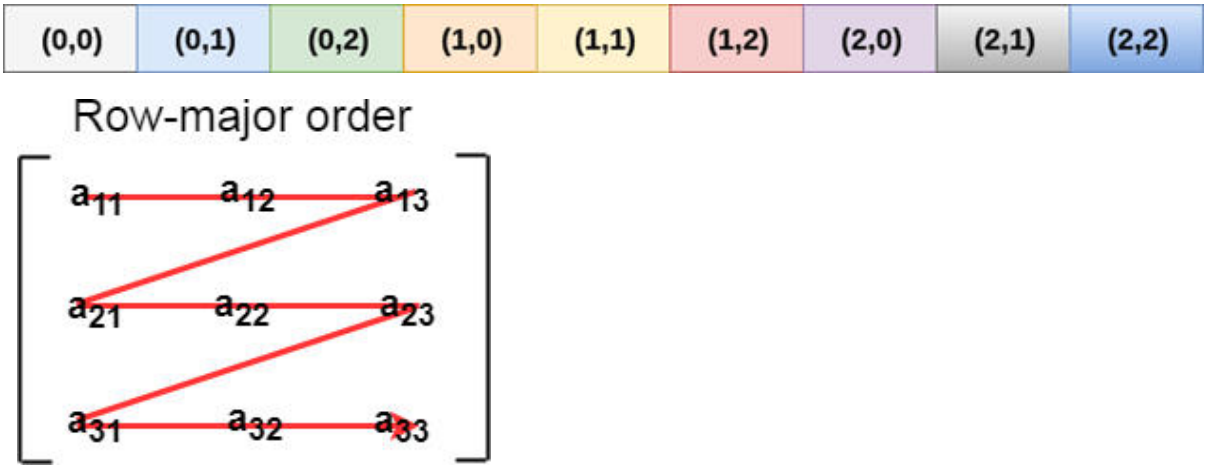


There are two main techniques of storing 2D array elements into memory

1. Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

first, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.



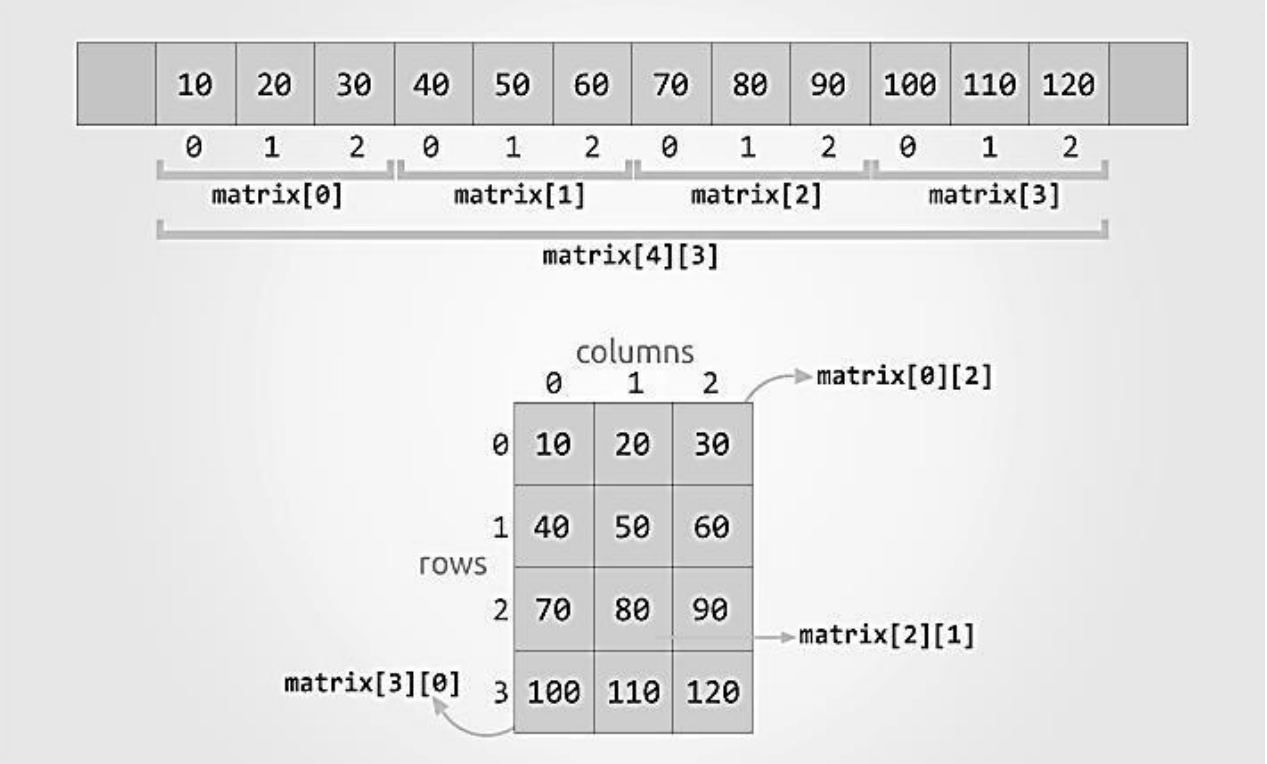
2. Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in in the above image is given as follows.

first, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.

Two-dimensional array

Two-dimensional array is a collection of one-dimensional array. Two-dimensional array has special significance than other array types. You can logically represent a two-dimensional array as a matrix. Any matrix problem can be converted easily to a two-dimensional array.



Syntax to declare two-dimensional array

type array_name[row-size][col-size];

- ✓ type is a valid C data type.
- ✓ array_name is a valid C identifier that denotes name of the array.
- ✓ row-size is a constant that specifies matrix row size.
- ✓ col-size is also a constant that specifies column size. col-size is optional when initializing array during its declaration.

Example to declare two-dimensional array

int matrix[3][4];

The above statement declares a two-dimensional integer array of size 3x4 i.e. 3 rows and 4 columns (in terms of matrix).

How to initialize two-dimensional array

You can initialize a two-dimensional array in any of the given form.

```
int matrix[4][3] = {
                        {10, 20, 30},    // Initializes matrix[0]
                        {40, 50, 60},    // Initializes matrix[1]
                        {70, 80, 90},    // Initializes matrix[2]
                        {100, 110, 120}  // Initializes matrix[3]
                    };
```

Basic Operations

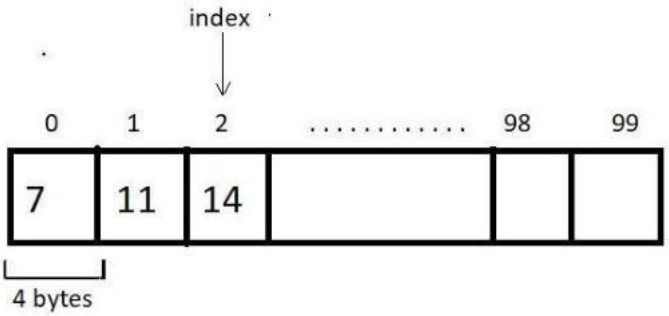
Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

Traversal:

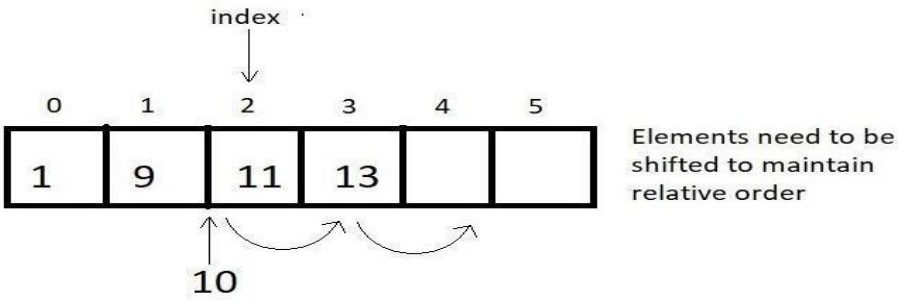
Visiting every element of an array once is known as traversing the array.
For use cases like:
Storing all elements – Using scanf()
Printing all elements – Using printf()

Updating elements.
An array can easily be traversed using a for loop in C language.



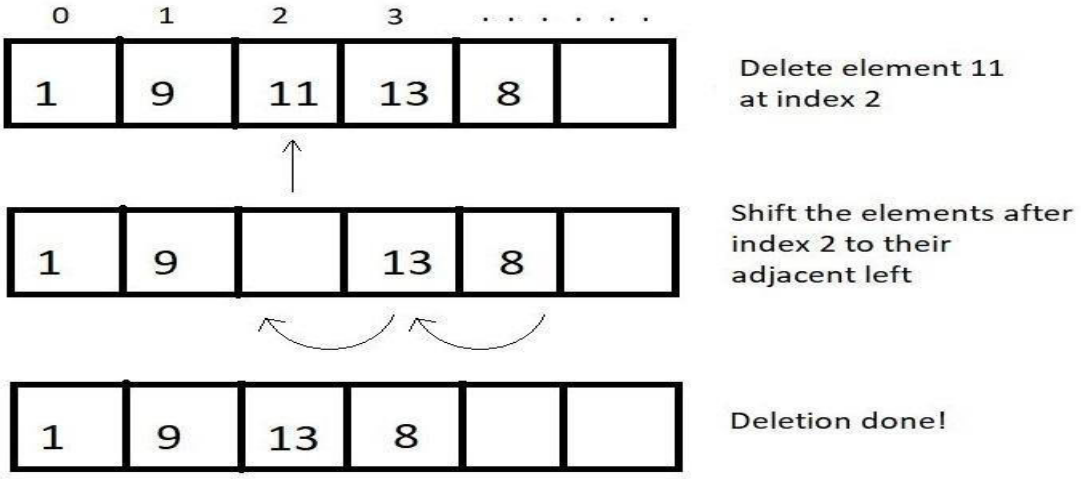
Insertion:

An element can be inserted in an array at a specific position. For this operation to succeed, the array must have enough capacity. Suppose we want to add an element 10 at index 2 in the below-illustrated array, then the elements after index 1 must get shifted to their adjacent right to make way for a new element.



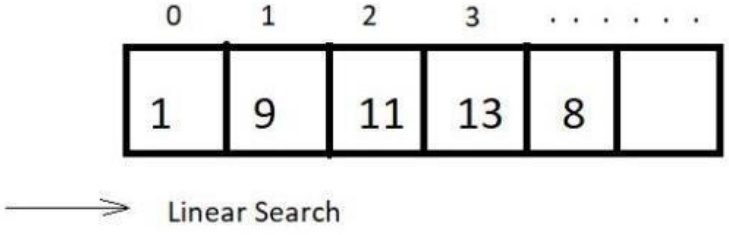
Deletion:

An element at a specified position can be deleted, creating a void that needs to be fixed by shifting all the elements to their adjacent left, as illustrated in the figure below.



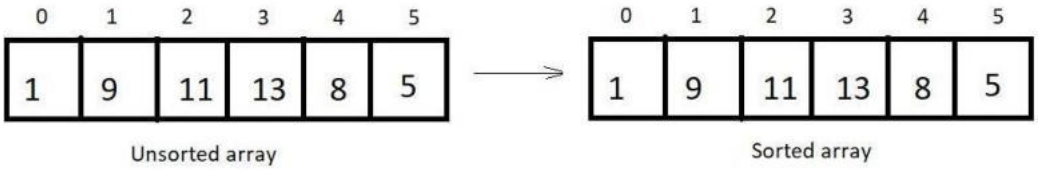
Searching:

Searching can be done by traversing the array until the element to be searched is found.O(n) There is still a better method.



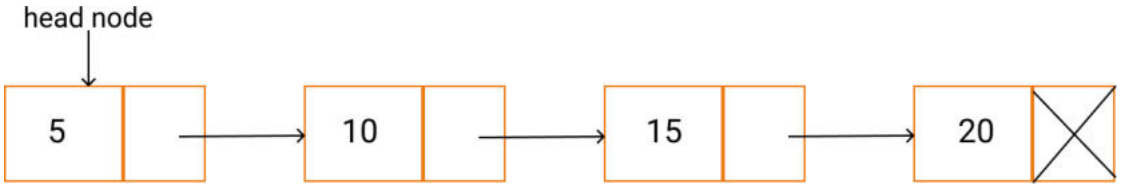
Sorting:

Sorting means arranging an array in an orderly fashion (ascending or descending). We have different algorithms to sort arrays.



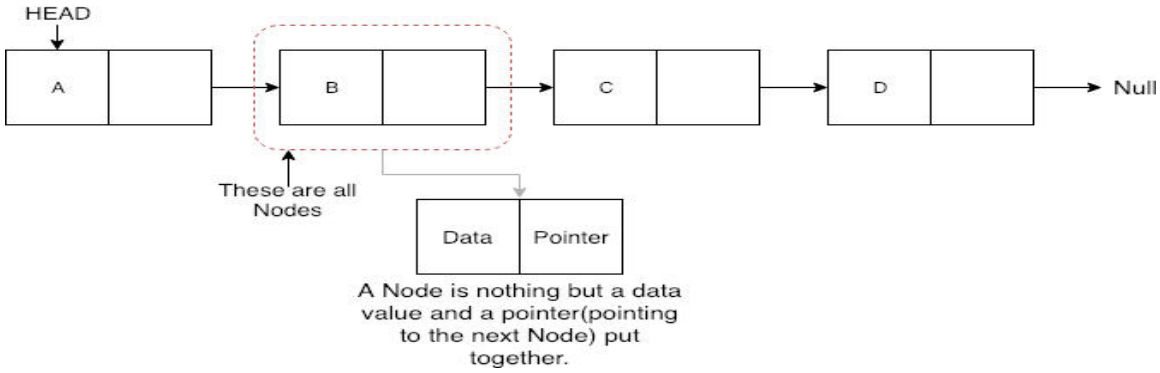
Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.
- Each node is composed of **data** and a **reference to the next node** in the sequence. The last node has a reference to null which indicates the end of the linked list.
- Head node is the starting node of the linked list(first node) and it contains the reference to the next node in the list. The head node will have a null reference when the list is empty.



What is a Node?

A Node in a linked list holds the data value and the pointer which points to the location of the next node in the linked list.



Why Linked lists?

Arrays and Linked Lists are both linear data structures, but they both have some advantages and disadvantages over each other.

- One advantage of the linked lists is that elements can be added to it indefinitely without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory. while an array will eventually get filled or have to be resized for making an insertion whenever needed (a costly operation that isn't always possible).
- Elements are also be easily removed from linked lists whereas removing elements from an array leaves empty spaces that are a waste of computer memory or performing a shift operation in arrays increases the cost thus makes it expensive.

Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked List

Following are the various types of linked list.

a) **Simple Linked List** − Item navigation is forward only.

b) **Doubly Linked List** − Items can be navigated forward and backward.

c) **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.

Representation of Linked Lists

a) Singly Linked Lists

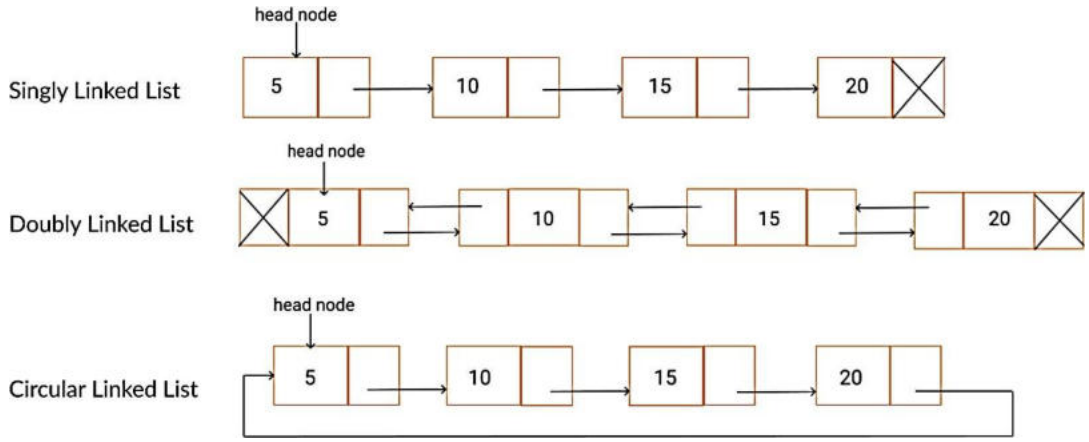
- The first node is the head node and it points to next node in the sequence.
- The last node's reference is null indicating the end of the list.

b) Doubly Linked Lists

- Every node has two pointers, one for pointing to next node and the other for pointing to the previous node.
- The **next** pointer of the last node and the previous pointer of the first node (head) are null.

c) Circular Linked Lists

- Circular Linked List is very similar to a singly linked list except that, here the last node points to the first node making it a circular list as shown below.



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.

Singly linked list

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list.

Insertion

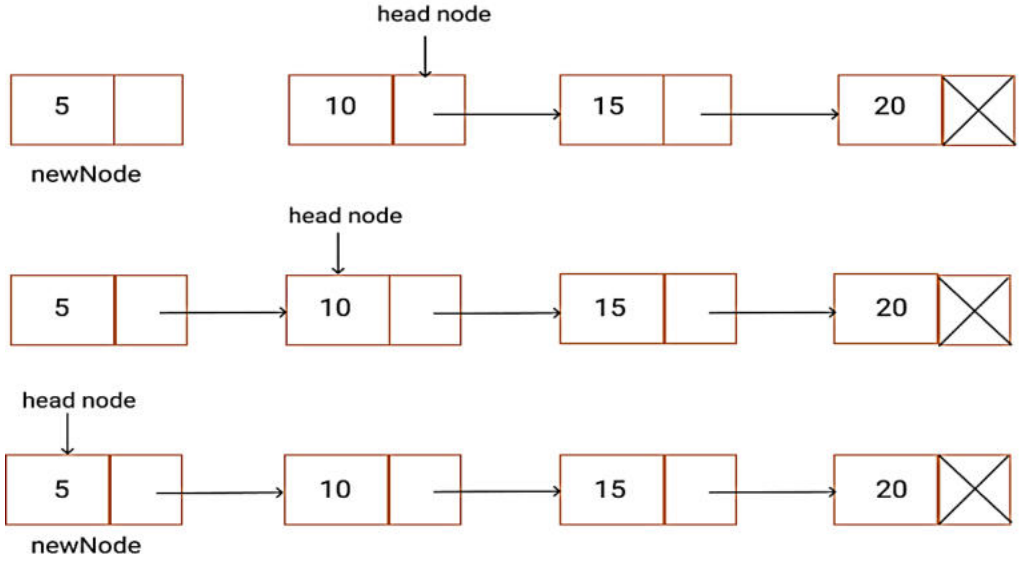
The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Insertion in Linked List can happen at various places in a Linked List. A few cases are:

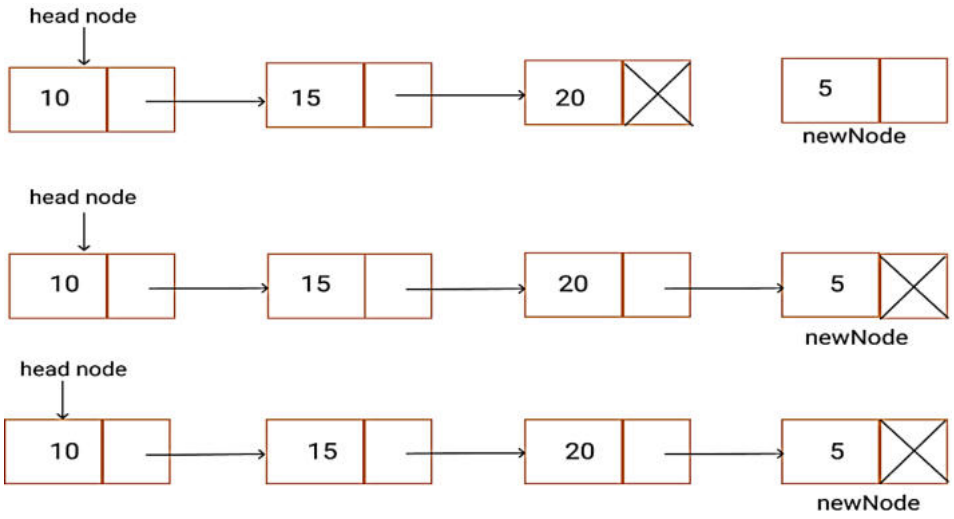
- At the beginning of the linked list.
- At the middle of the linked list.
- At the end of the linked list.
- At a given position in the linked list.

Insertion at the beginning of the Linked List



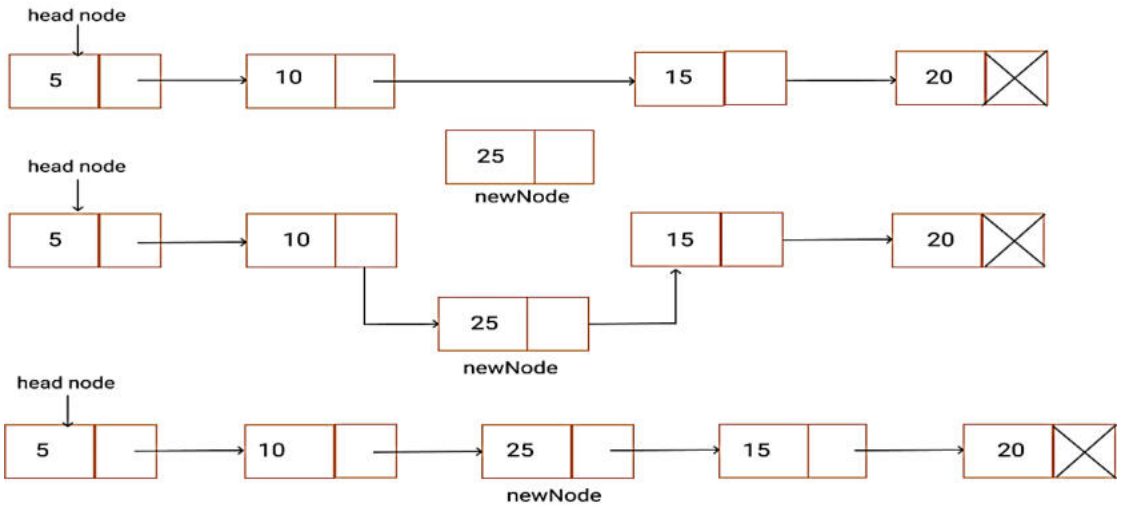
- Let us assume a newNode as shown above. The **newNode** with data=5 has to be inserted at the beginning of the linked list.
- For this to happen, the address part of the **newNode** should point to the address of the head node. Once it points to the head node, then the **newNode** is made as the head node as shown above.

Insertion at the end of the linked list



- Now let us assume a newNode as shown above. The **newNode** with data = 5 has to be inserted at the end of the linked list.
- Now, start traversing the list and continue traversing until the address part of the node is NULL.
- Once the address part of a node is found to be NULL, then make the address part of that node to point to the address of the **newNode** and the address part of the **newNode** is referenced to NULL. Hence it becomes the last node.

Insertion at a given position in the linked list



- Now let us assume that a newNode with data = 25 has to be inserted at the 2nd position in the linked list.
- Start traversing the list from the head and move up to the **position - 1.** In this case, since we want to insert at 2nd position, you need to traverse till (2-1) = 1.
- Once you reach the **position-1** node, the address part of the newNode is set to the address contained by the **position – 1** node.
- Now, the address part of the **position – 1** node is made to point to the newNode.

Deletion and Traversing

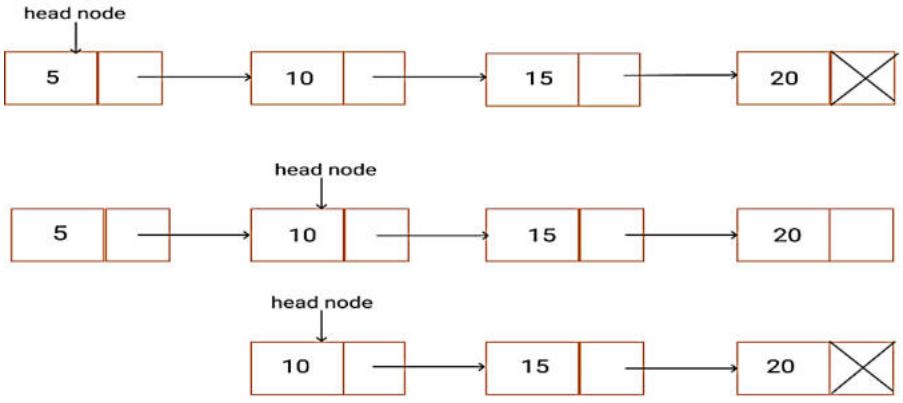
The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Deletion in a linked list can happen at various places in a list. A few cases are:

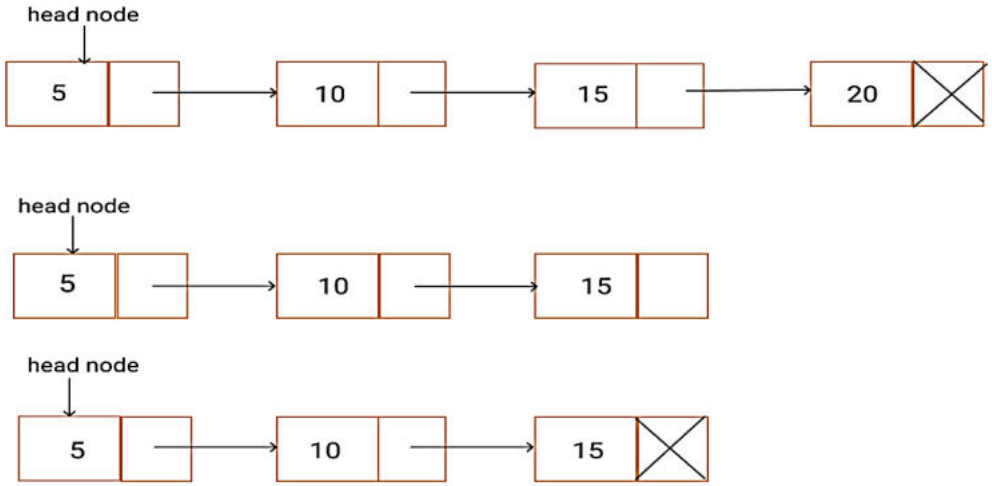
- At the beginning of the linked list.
- At the end of the linked list.
- At a given position in the linked list.

Deletion at the beginning of the linked list



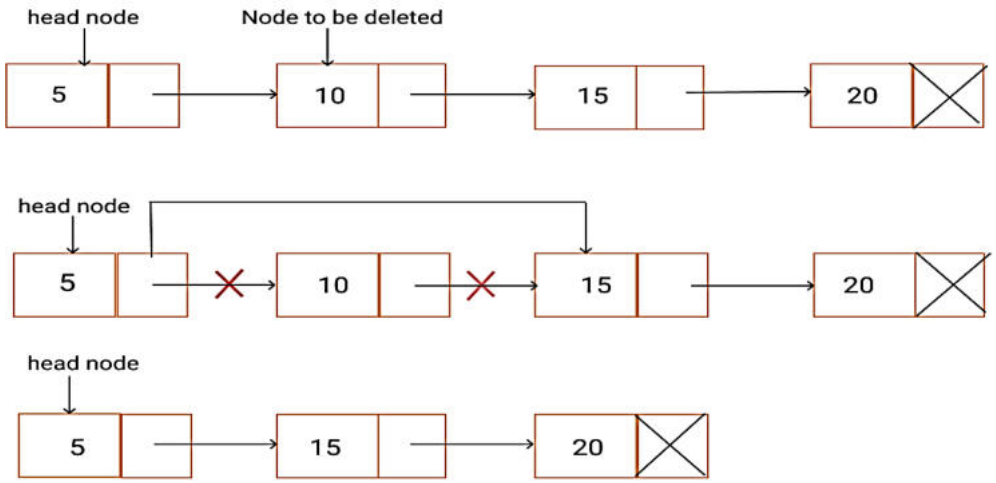
- To delete the first node (head node), copy the head node in a temporary node.
- Make the second node as the head node.
- Now, delete the temporary node.

Deletion at the end of the linked list



- To delete the last node, start traversing the list from the head node and continue traversing until the address part of the node is **NULL**.
- Keep track of the **second last node** in some temporary variable say **prev_node**.
- Once the address part of the node is NULL, set the **address part** of the **prev_node** as **NULL** and then delete the last node.

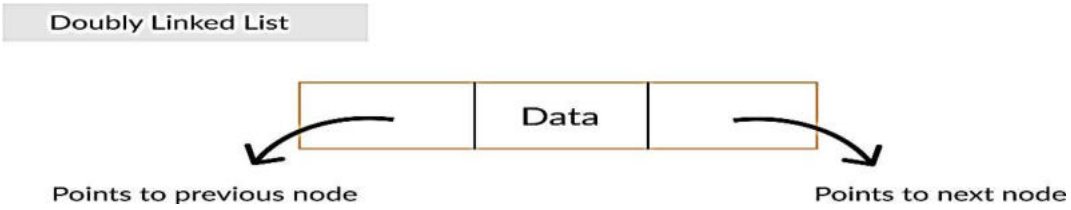
Deletion at a given position in the linked list



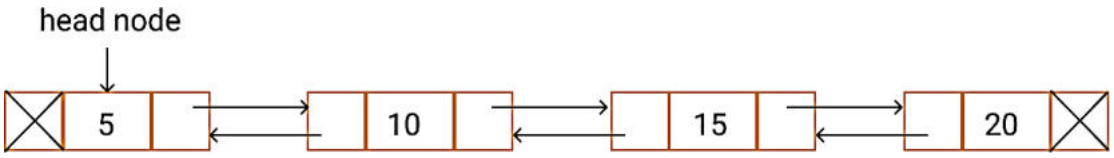
- Now let us assume that the node at position 2 has to be deleted.
- Start traversing the list from the head node and move up to that **position.**
- While traversing, keep track of the previous node to the node to be deleted.
- In this case, since we want to delete the second node, you need to traverse till node 2, storing node 1 in some temporary variable.
- Now, the address part of node 2 is assigned to the address part of node 1 and then node 2 is deleted.

Doubly Linked List

Unlike a Singly Linked List, in a Doubly Linked List, there is an extra pointer, the **previous** pointer along with the **next** pointer as shown below. The previous pointer points to the previous node in the list and the next pointer points to the next node in the list. So, in a doubly linked list, a node contains a reference both to its previous node and to its next node.



Representation



Implementation of Doubly Linked List

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Basic operations
The following are the basic operations supported by this Linked List.

- **Insertion** − Inserts an element at a specified position in the list.
- **Deletion** − Deletes an element at a specified position in the list.
- **Search** − Searches an element using the given value.
- **Update** − Updates/modifies an element in the list with the given value.
- **Reverse** − Reverses the entire Linked list.
- **Traverse** − Displays all the elements in the linked list.

Advantages and Disadvantages of Doubly Linked List over Singly Linked List

- Traversal can be done in both directions (from the start node to the end node as well as from the end node to the start node) in a Doubly Linked list. But this is not possible in a Singly Linked List and it can only be traversed only in one direction.
- Deletion and insertion operations are easy to implement in a Doubly LL than a Singly LL. For example, in a singly linked list, to delete a node, the pointer to the previous node is needed for which the list is to be traversed. In a Doubly LL, we just need to know the pointer of the node to be deleted.
- Memory has to be allocated for both the next and previous pointers in a node. Hence, the occupation of memory is higher in Doubly LL.
- Both the pointers will have to be modified if any kind of operation is performed like insertion, deletion, etc in case of Doubly LL.

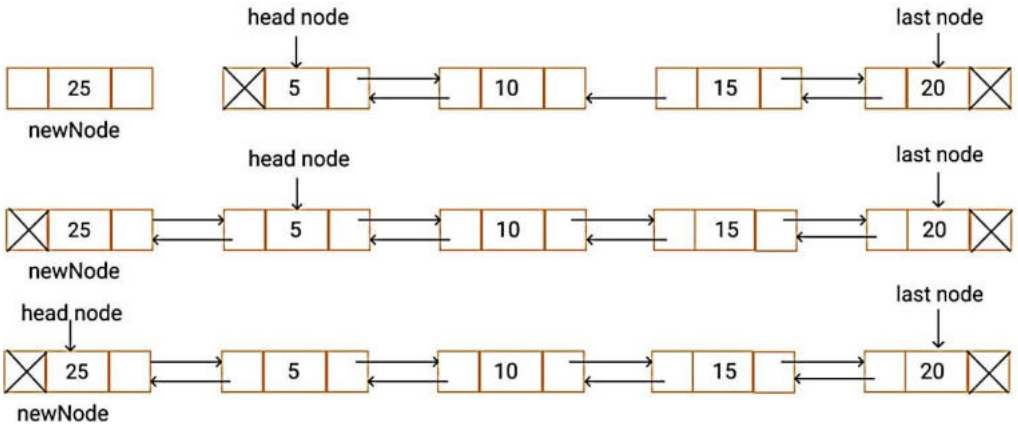
Applications

- The browser cache which allows you to hit the BACK buttons or navigate through previous pages.
- Applications that have a Most Recently Used (MRU) lists.
- A stack, hash table, and binary tree can be implemented using a doubly linked list.
- Undo features in publishing or editing applications like Photoshop and Word.
- A great way to represent a deck of cards in a game where removing items from anywhere in the deck is essential.

Operations on doubly linked list
Insertion in doubly linked list can happen at various places in the list. A few cases are:

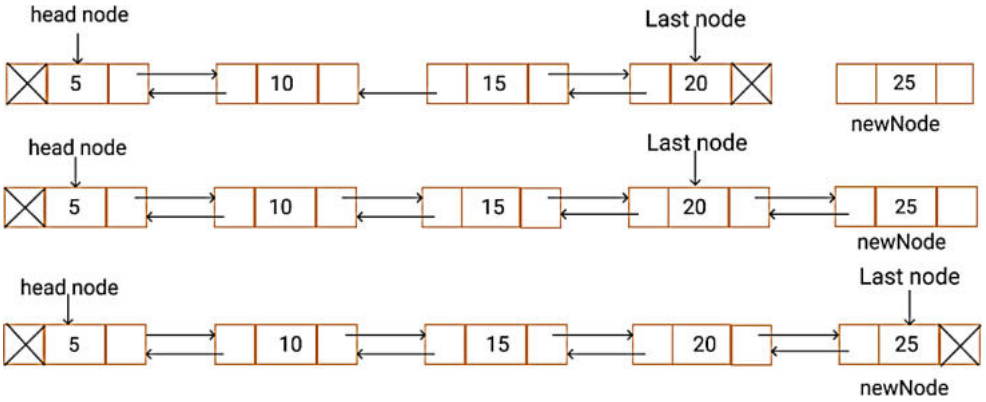
- At the beginning of the doubly linked list.
- At the end of the doubly linked list.
- At a given position in the doubly linked list.

Insertion at the beginning of the doubly linked list



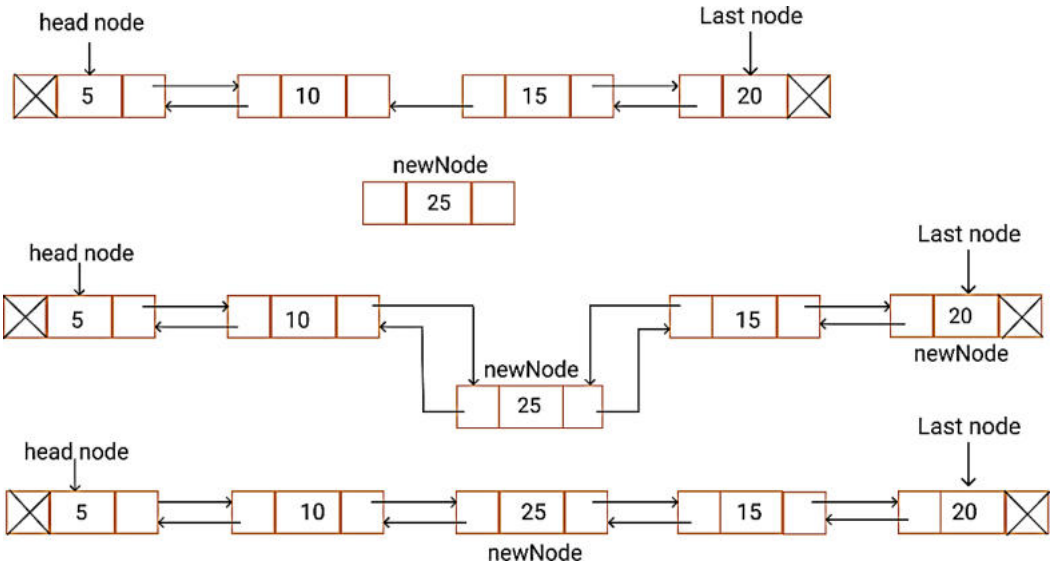
- Let us assume a newNode as shown above. The newNode with data = 25 has to be inserted at the beginning of the list.
- The **next** pointer of the **newNode** is referenced to the head node and its **previous** pointer is referenced to **NULL**.
- The **previous** pointer of the head node is referenced to the **newNode**.
- The **newNode** is then made as the head node.

Insertion at the end of the doubly linked list



- Now, let us assume a newNode as shown above. The **newNode** with data = 25 has to be inserted at the end of the linked list.
- Make the **next** pointer of the **last** node to point to the **newNode** .
- The **next** pointer of the **newNode** is referenced to NULL and its **prev** pointer is made to point to the last node.Then, the **newNode** is made as the **last node**.

Insertion at the given position in the doubly linked list



- Now let us assume that a **newNode** with data = 25 has to be inserted at position 2 in the linked list.

- Start traversing the list from head and move upto the **position - 1**. In this case, since we want to insert at second position, you need to traverse till (2 - 1) = 1 st node.
- Once you reach the position - 1 node, the **next** pointer of the newNode is set to the address contained by the **next** pointer of the (position - 1) th node.
- Make the **next** pointer of the (position - 1) th node to point to the newNode and the prev pointer of the newNode is pointed to the (position - 1) th node
- Finally the newNode next 's **prev** pointer must be made to point to the newNode.

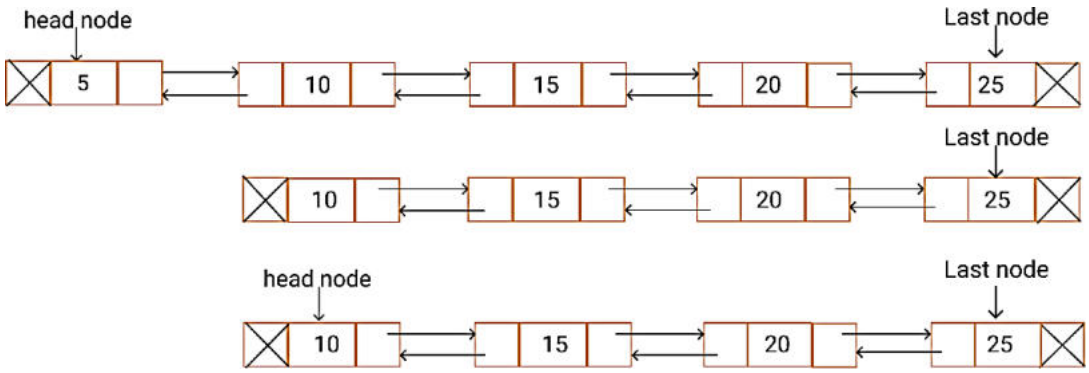
Deletion in doubly linked list

Deletion in a doubly linked list can happen at various places in the list.

A few cases are:

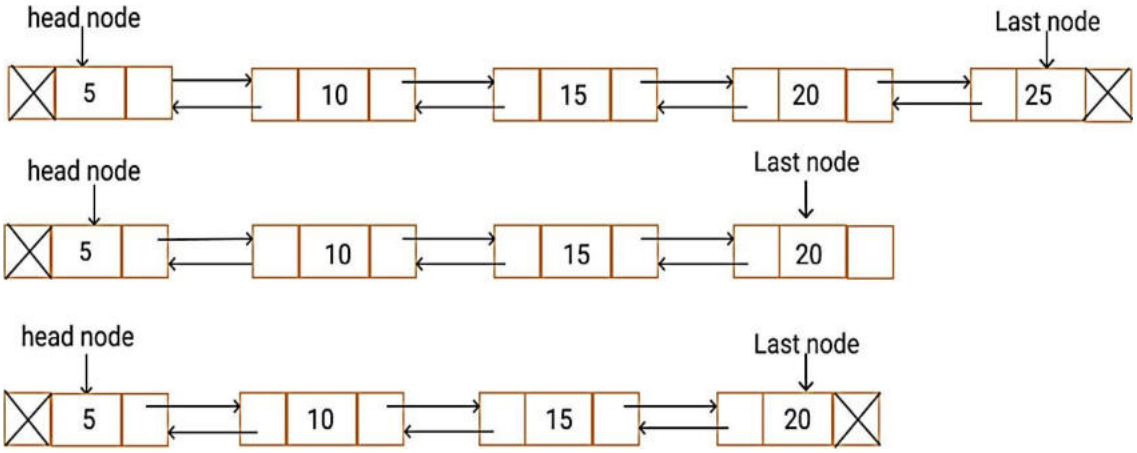
- At the beginning of the doubly linked list.
- At the end of the doubly linked list.
- At a given position in the doubly linked list.

Deletion at the beginning of the doubly linked list



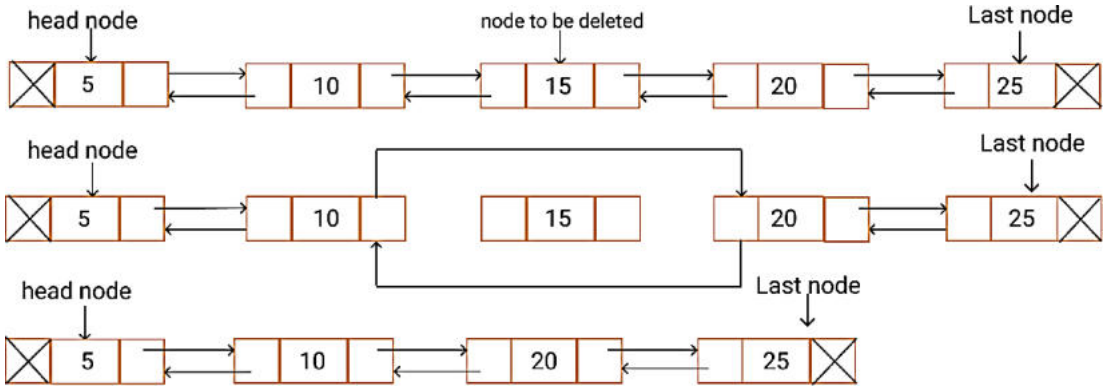
- Copy the head node in some temporary node.
- Make the second node as the head node.
- The **prev** pointer of the head node is referenced to NULL.
- Delete the temporary node.

Deletion at the end of the doubly linked list



- Copy the **last** node to a temporary node.
- Shift the **last** node to the second last position.
- Make the **last** node's **next** pointer as NULL.
- Delete the temporary node.

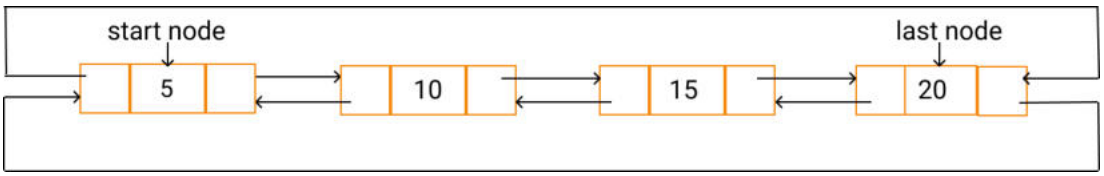
Deletion at the given position in the doubly linked list



- Suppose you want to delete the second node from the list.
- Start traversing the linked list from the head until the position = 2 of the node to be deleted.
- Let the node at the position 2 of the list be **temp**.
- Assign the **next** pointer of temp to temp's previous node's **next** pointer.
- Assign the temp's **prev** pointer to temp's next node's **prev** pointer.
- Delete the **temp** node.

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.



Advantages of Circular Linked Lists

- Traversal can begin from any node in the list. When the started node is visited again, the list is meant to complete one cycle.
- Used in Queue data structure implementation.
- Circular Doubly Linked Lists are used for implementation of Fibonacci Heaps.

Basic Operations in Circular Linked lists

The following basic operations can be done in both singly and doubly circular linked lists.

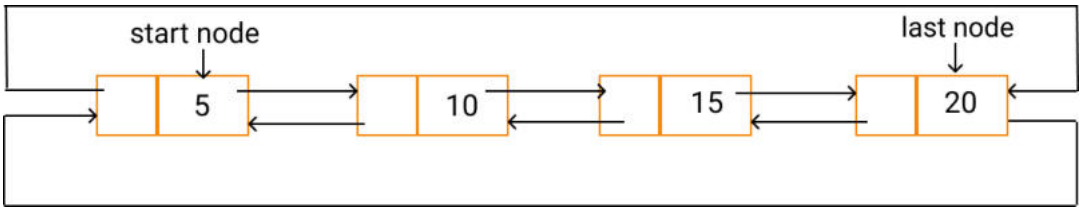
- **Insertion** − Inserts an element at specified positions in the list.
- **Deletion** − Deletes the specified element from the list.
- **Search** − Searches an element using the given value.
- **Update** − Update an element in the list with the given value.
- **Reverse** − Reverses the entire Linked list.
- **Traverse** − Displays all the elements in the linked list.

Applications of Circular Linked Lists

- Circular Linked Lists can be used in any application where the entries appear in a circular manner.
- Circular Linked Lists is the basic idea for the round robin scheduling algorithm.
- Circular Linked Lists are used majorly in time-sharing applications where the operating system maintains a list of current users and must alternately allow each user to use a small amount of CPU time (one user at a time). The operating system will pick a user, allow him/her use a small amount of CPU time and then move to the next user.

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.



The following basic operations can be done in a Circular Singly Linked List

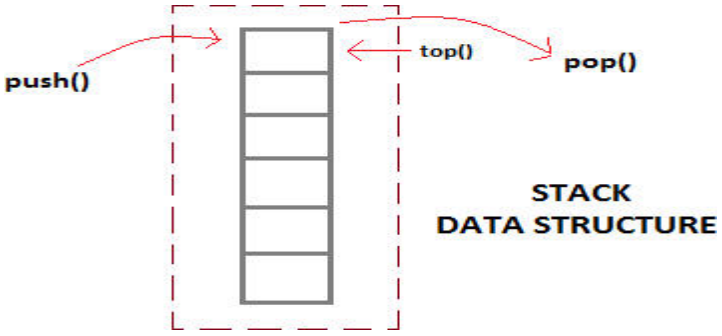
- **Insertion** − Inserts an element at specified positions in the list.
- **Deletion** − Deletes the specified element from the list.
- **Search** − Searches an element using the given value.
- **Update** − Update an element in the list with the given value.
- **Reverse** − Reverses the entire Linked list.
- **Traverse** − Displays all the elements in the linked list.

What is a Stack?

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. A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.

It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the

element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***



Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. 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**.
4. 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.

The practical examples of stacks are:

- Arrangement of plates.
- Arrangement of books in a cupboard.
- Wearing or removing bangles from hands.

Standard Stack Operations

The following are some common operations implemented on the stack:

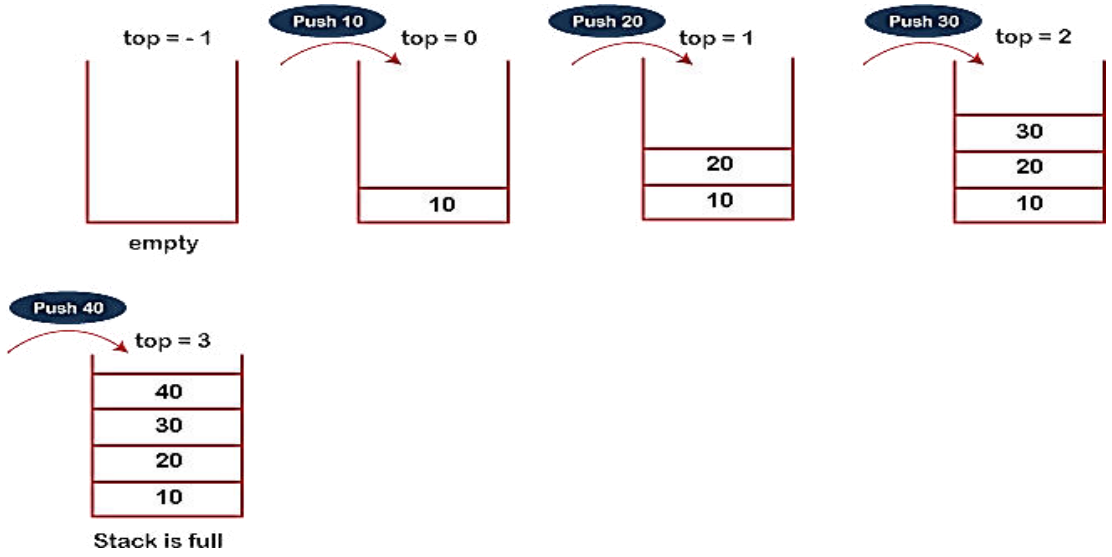
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.

- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.

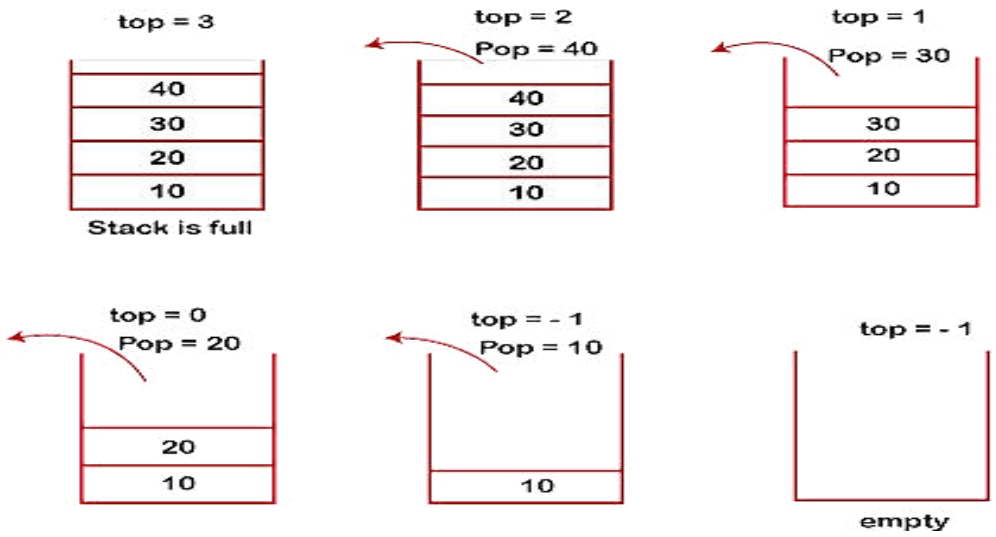


POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***

- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Applications of Stack

String reversal: **Stack is also used for reversing a string.**

UNDO/REDO: **It can also be used for performing UNDO/REDO operations.**

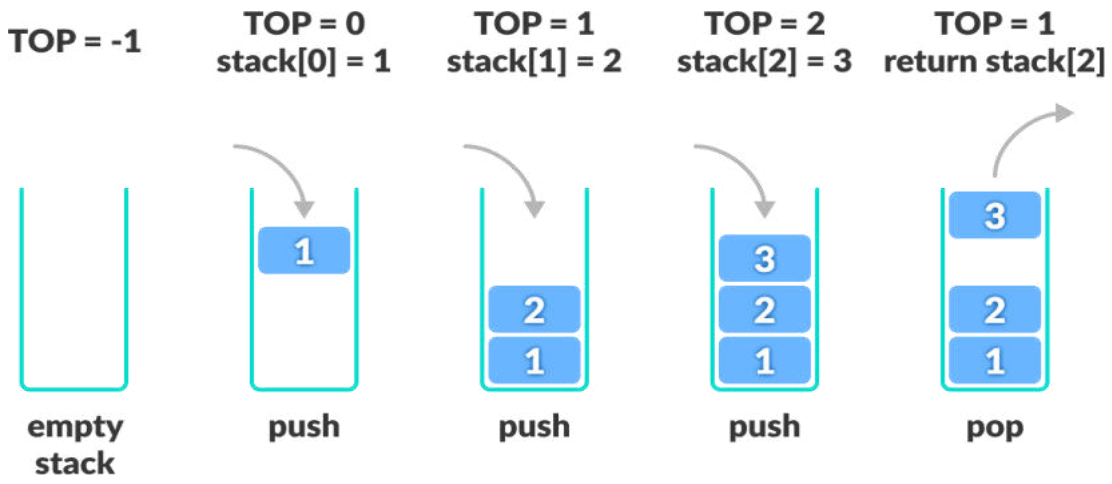
Recursion: **The recursion means that the function is calling itself again.**

Expression conversion: Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
Infix to prefix
Infix to postfix
Prefix to infix
Prefix to postfix
Postfix to infix

Working of Stack Data Structure

The operations work as follows:

1. A pointer called TOP is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. On popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty



Representation of stacks
Stacks can be represented using both arrays and linked lists.

Array implementation of Stack

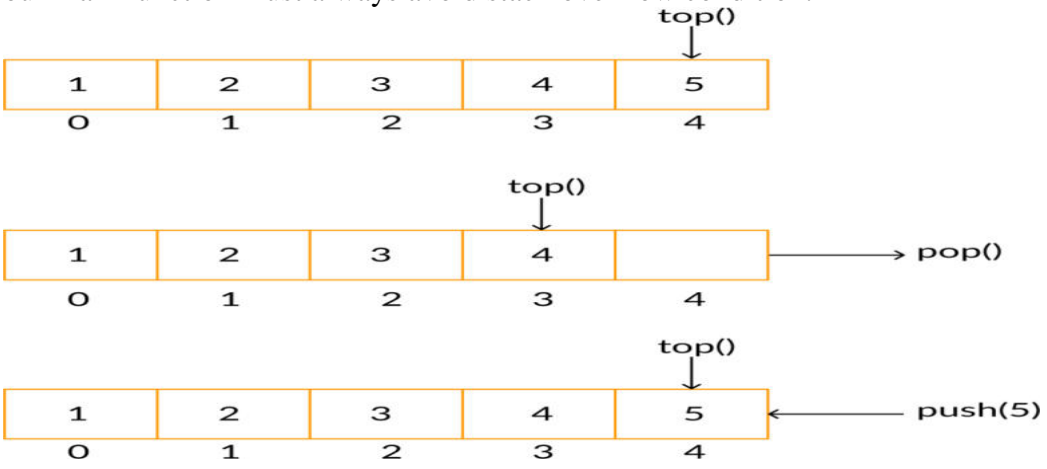
In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

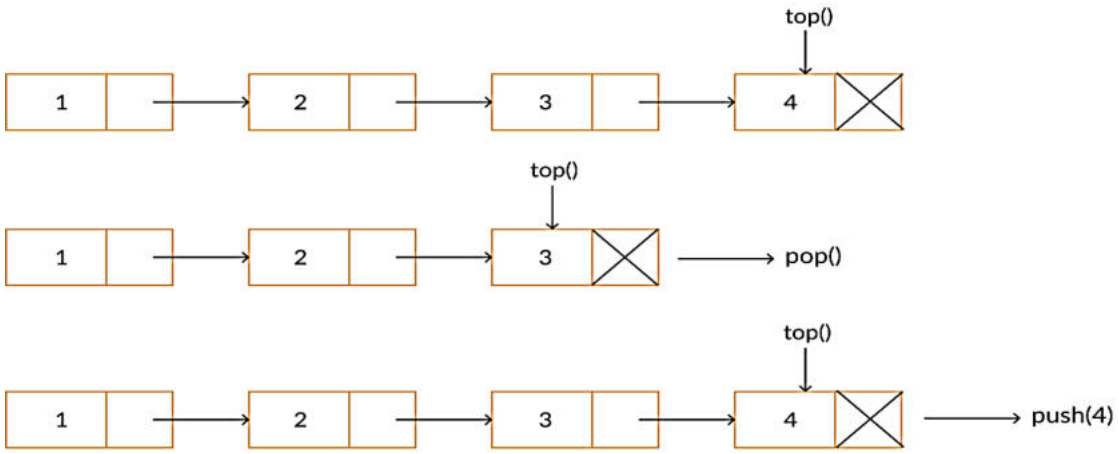


1. Initially, the topmost element from the array is **5**.
2. Performing a pop operation, deletes the topmost element 5 from the array and makes 4 as the topmost element.
3. Now, on performing push operation to add element 5, the element is added and is made as the topmost element.

Linked list implementation of 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 overflown if the space left in the memory heap is not enough to create a node.



1. Initially, the topmost element in the linked list is **4**.
2. Performing a pop operation, deletes the topmost element **4** from the list and makes **3** as the topmost element.
3. Now, on performing push operation to add element 4, the element is added and is made as the topmost element.

Stack applications

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

1. Stacks can be used for expression evaluation.
2. Stacks can be used to check parenthesis matching in an expression.
3. Stacks can be used for Conversion from one form of expression to another.
4. Stacks can be used for Memory Management.
5. Stack data structures are used in backtracking problems.

Expression Evaluation

Stack data structure is used for evaluating the given expression. For example, consider the following expression

5 * (6 + 2) - 12 / 4

Since parenthesis has the highest precedence among the arithmetic operators, **(6 +2) = 8** will be evaluated first. Now, the expression becomes

5 * 8 - 12 / 4

* and / have equal precedence and their associativity is from left-to-right. So, start evaluating the expression from left-to-right.

5 * 8 = 40 and **12 / 4 = 3**

Now, the expression becomes

40 - 3

And the value returned after the subtraction operation is **37**.

Parenthesis Matching
Given an expression, you have to find if the parenthesis is either correctly matched or not. For example, consider the expression **(a + b) * (c + d).**

In the above expression, the opening and closing of the parenthesis are given properly and hence it is said to be a correctly matched parenthesis expression. Whereas, the expression, **(a + b * [c + d)** is not a valid expression as the parenthesis are incorrectly given.

Expression Conversion
Converting one form of expressions to another is one of the important applications of stacks.

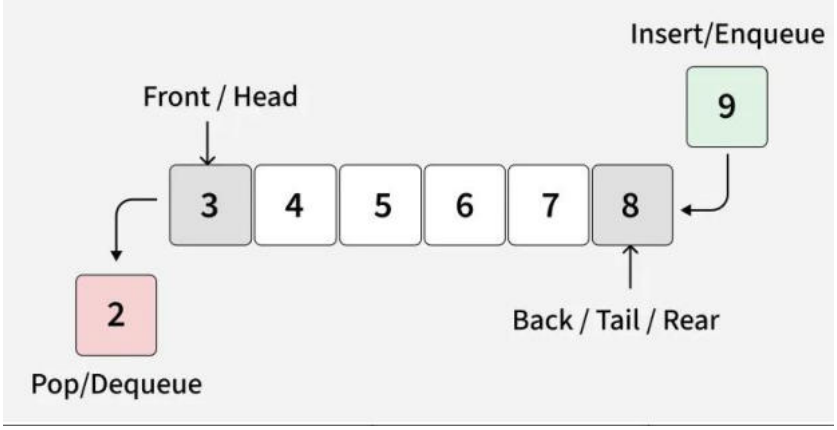
- Infix to prefix
- Infix to postfix
- Prefix to Infix
- Prefix to Postfix
- Postfix to Infix
- Postfix to Infix

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

Queues

Queue is a linear data structure that follows **FIFO** (First In First Out) Principle, so the first element inserted is the first to be popped out.

FIFO Principle in Queue:
FIFO Principle states that the first element added to the Queue will be the first one to be removed or processed. So, Queue is like a line of people waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).



Basic Terminologies of Queue

- **Front:** Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue. It is also referred as the head of the queue.
- **Rear:** Position of the last entry in the queue, that is, the one most recently added, is called the rear of the queue. It is also referred as the tail of the queue.
- **Size:** Size refers to the current number of elements in the queue.
- **Capacity:** Capacity refers to the maximum number of elements the queue can hold.

Types of Queues

1. Simple Queue

A simple queue follows the FIFO (First In, First Out) principle.

- Insertion is allowed only at the rear (back).
- Deletion is allowed only from the front.
- Can be implemented using a linked list or a circular array.

When an array is used, we often prefer a **circular queue**, which is mainly an efficient array implementation of a simple queue. It efficiently utilizes memory by reusing the empty spaces left after deletion, avoiding wastage that occurs in a normal linear array implementation.

2. Double-Ended Queue (Deque)

In a deque, insertion and deletion can be performed from both ends.

3. Priority Queue

A queue where each element is assigned a **priority**, and deletion always happens based on priority (not just position).

Queue Operations

1. **Enqueue**: Adds an element to the end (rear) of the queue. If the queue is full, an overflow error occurs.
2. **Dequeue**: Removes the element from the front of the queue. If the queue is empty, an underflow error occurs.
3. **Peek/Front**: Returns the element at the front without removing it.
4. **Size**: Returns the number of elements in the queue.
5. **isEmpty**: Returns true if the queue is empty, otherwise false.
6. **isFull**: Returns true if the queue is full, otherwise false.

Applications of Queues

1. CPU and Disk Scheduling

- Processes are scheduled in order of arrival using queues (like Round Robin Scheduling).

2. Breadth-First Search (BFS)

- Uses queue to explore nodes level by level in graphs or trees.

3. Handling Asynchronous Data

- Queues buffer data in real-time systems like I/O Buffers, streaming, or network data.

4. Resource Sharing and Task Scheduling

- Queues are used in operating systems to manage threads, jobs, or resources.

5. Messaging Services

- In applications like RabbitMQ, Kafka, etc., queues manage message delivery between producers and consumers.

6. Simulation Systems

- Queues are used to simulate real-world queues for study or prediction (e.g., airport check-ins).

7. Memory Management

- Used in garbage collection algorithms like reference counting.

UNIT - II

Non Linear Data Structures: The data structure where data items are not organized sequentially (hierarchical) is called non linear data structure.

Nonlinear data structures are those data structures in which data items are not arranged in a sequence.

Examples of Non-linear Data Structure are Tree and Graph.

***Tree*:** A tree can be defined as a finite set of data items (nodes) in which data items are arranged in branches and sub-branches according to requirements.

Trees represent the hierarchical relationship between various elements

Tree consists of nodes connected by an edge, the node represented by a circle and edge lives connecting to the circle.

***Graph*:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

A tree can be viewed as a restricted graph.

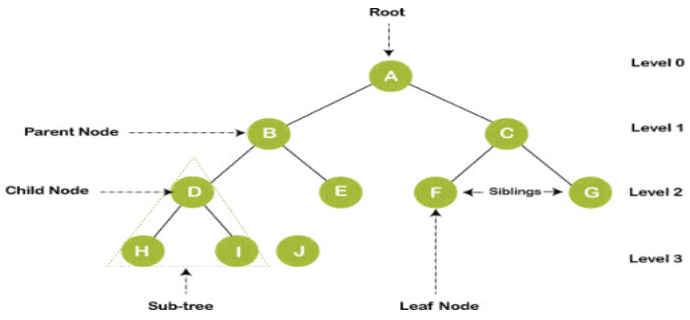
Graphs have many types:

- ✓ Un-directed Graph
- ✓ Directed Graph
- ✓ Mixed Graph
- ✓ Multi-Graph
- ✓ Simple Graph
- ✓ Null Graph
- ✓ Weighted Graph

Trees and **Graphs** are the types of non-linear data structure.

- **Tree**

It is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a **tree** data structure is shown below:

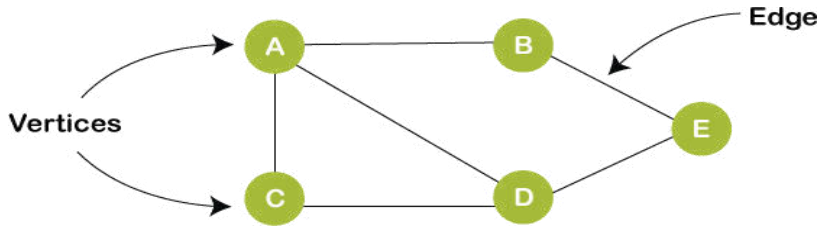


For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the above figure, **A** represents a manager, **B** and **C** represent the officers, and other nodes represent the clerks.

- **Graph**

A graph is a non-linear data structure that has a finite number of vertices and edges, and these

edges are used to connect the vertices. The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Facebook. In the case of facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.



	Linear Data structure	**Non-Linear Data structure**
Basic	In this structure, the elements are arranged sequentially or linearly and attached to one another.	In this structure, the elements **are arranged hierarchically or non-linear manner.**
Types	Arrays, linked list, stack, queue are the types of a linear data structure.	Trees and graphs are the types of a non-linear data structure.
implementation	Due to the linear organization, they are easy to implement.	Due to the non-linear organization, they are difficult to implement.
Traversal	As linear data structure is	The data items in a non-linear data
	a single level, so it requires a single run to traverse each data item.	structure cannot be accessed in a single run. It requires multiple runs to be traversed.
Arrangement	Each data item is attached to the previous and next items.	Each item is attached to many other items.
Levels	This data structure does not contain any hierarchy, and all the data elements are organized in a single level.	In this, the data elements are arranged in multiple levels.

Memory utilization	In this, the memory utilization is not efficient.	In this, memory is utilized in a very efficient manner.
Time complexity	The time complexity of linear data structure increases with the increase in the input size.	The time complexity of non-linear data structure often remains same with the increase in the input size.
Applications	Linear data structures are mainly used for developing the software.	Non-linear data structures are used in **image processing** and **Artificial Intelligence**.

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next time	Every item is attached with many other items.
Data is arranged in a linear sequence.	Data is not arranged in sequence
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Examples: Linked List, Stack, Queue, etc.	Examples: Trees, graphs, etc.
Implementation is easy.	Implementation is difficult.
Memory utilization Ineffective	Memory utilization Effective

Tree Data Structure-

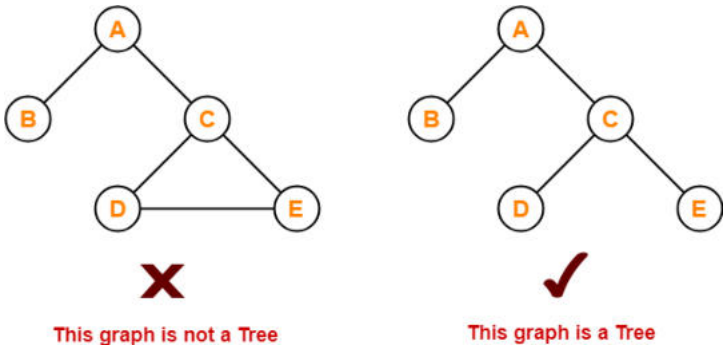
Tree data structure may be defined as-
Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.

OR

A tree is a connected graph without any circuits.

OR

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.



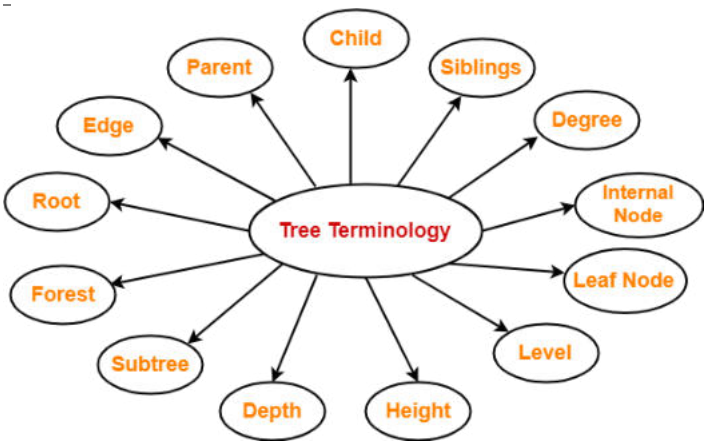
Properties-

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.

To gain better understanding about Tree Data Structure,

Tree Terminology-

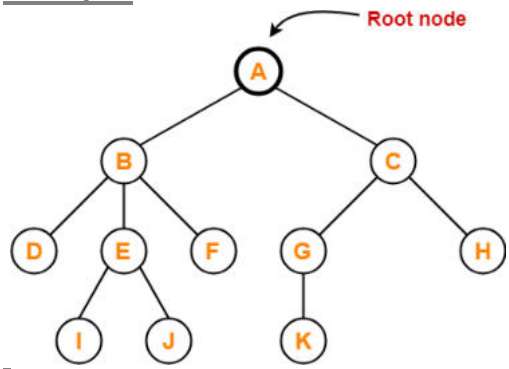


The important terms related to tree data structure are-

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.

Example-

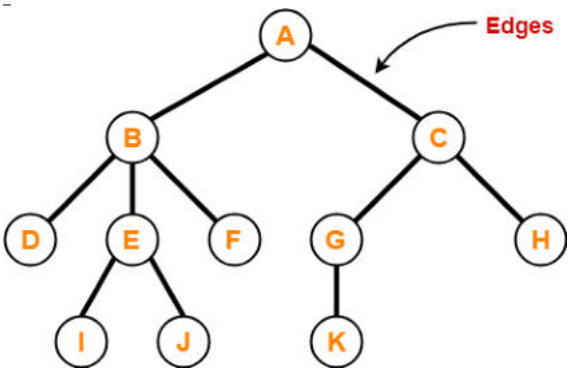


Here, node A is the only root node.

2. Edge-

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

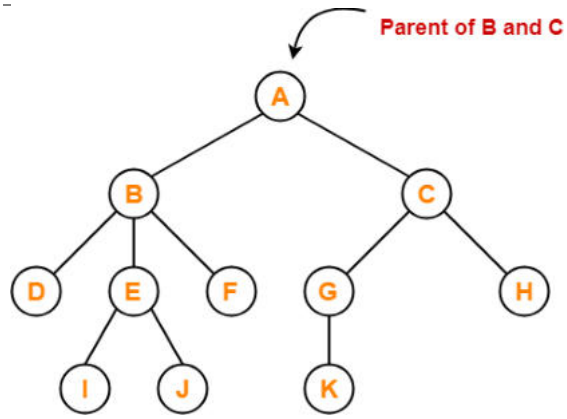
Example-



3. Parent-

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

Example-



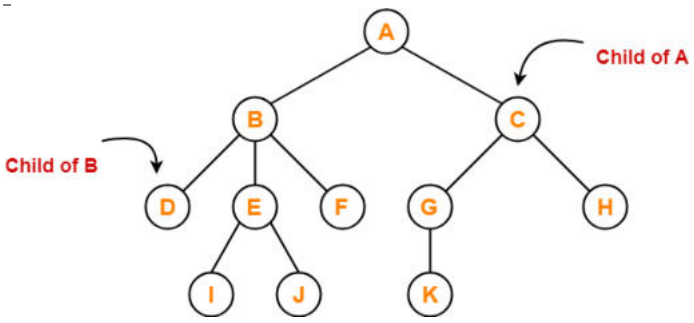
Here,

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

4. Child-

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

Example-



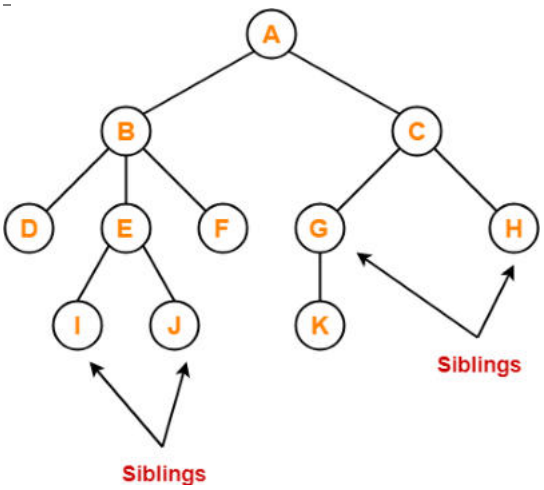
Here,

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

5. Siblings-

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

Example-



Here,

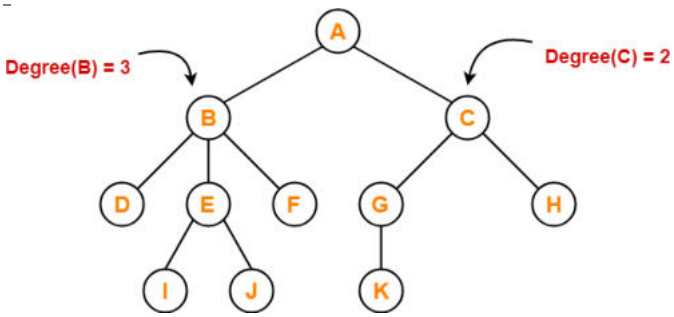
- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

6. Degree-

- **Degree of a node** is the total number of children of that node.

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

Example-



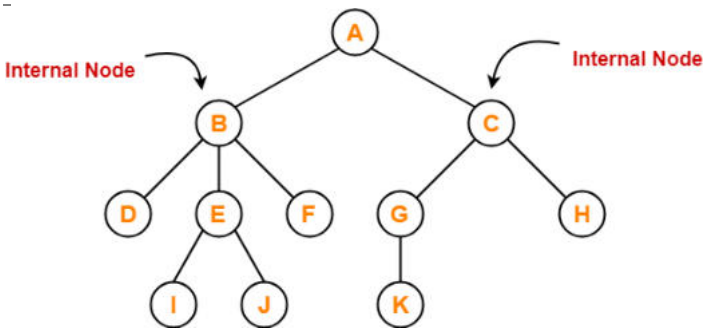
Here,

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

7. Internal Node-

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

Example-

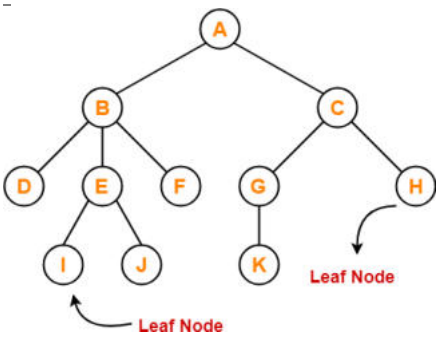


Here, nodes A, B, C, E and G are internal nodes.

8. Leaf Node-

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

Example-

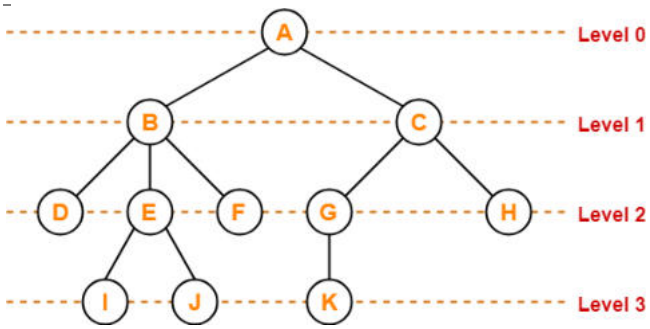


Here, nodes D, I, J, F, K and H are leaf nodes.

9. Level-

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

Example-



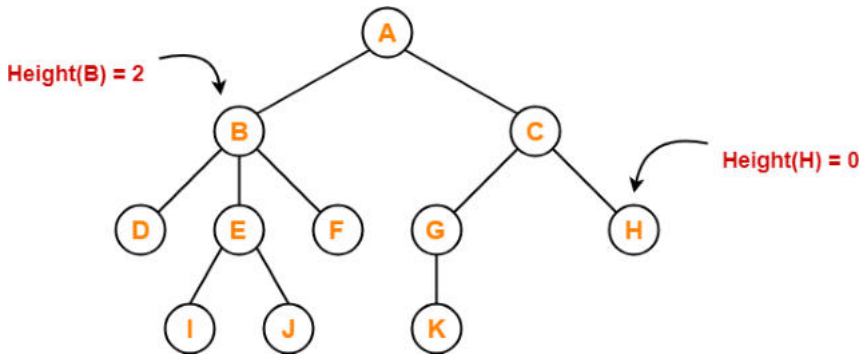
10. Height-

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

Example-

Here,

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1

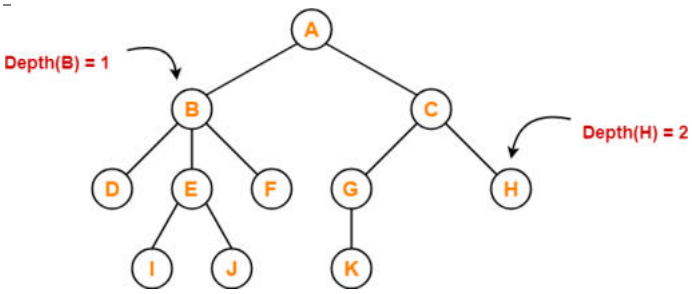


- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

11. Depth-

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms "level" and "depth" are used interchangeably.

Example-



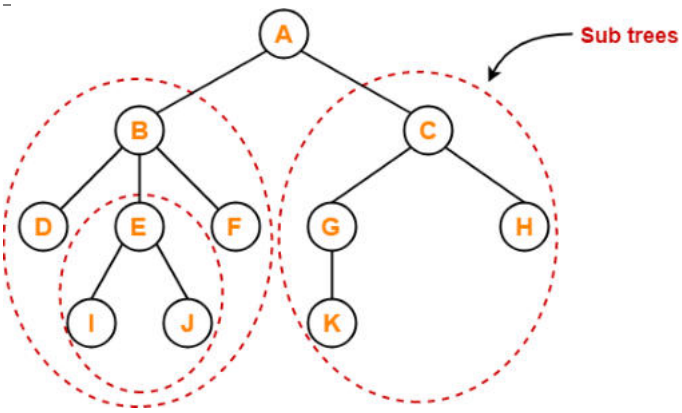
Here,

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

12. Subtree-

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

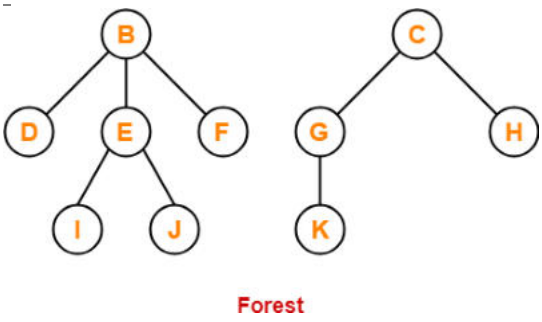
Example-



13. Forest-

A forest is a set of disjoint trees.

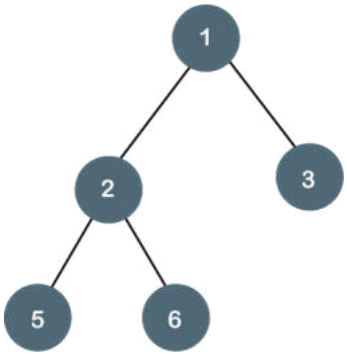
Example-

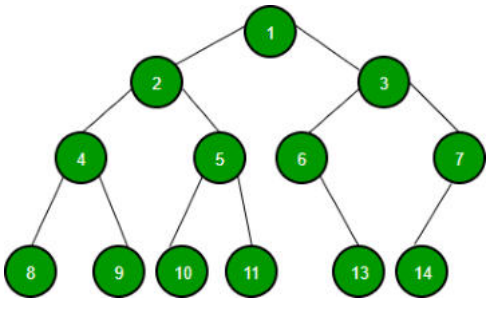


Forest

Binary Tree Data Structure

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.
(OR)
The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.





Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

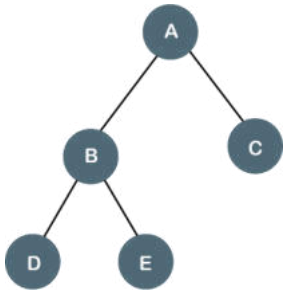
Properties of Binary Tree

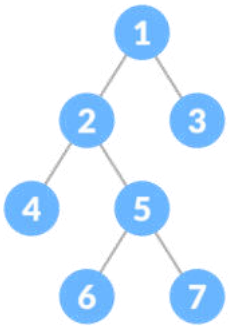
- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

Types of Binary Tree

Full Binary Tree

A full Binary tree is a special type of binary tree in which **every parent node/internal node has either two or no children.**

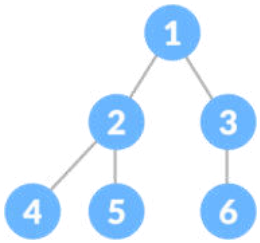


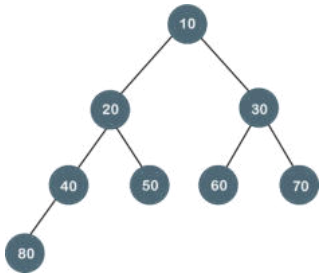


Complete Binary Tree

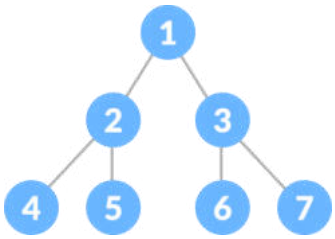
A complete binary tree is just like a full binary tree, but with two major differences

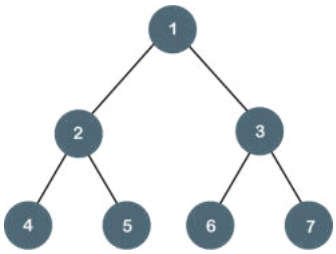
1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



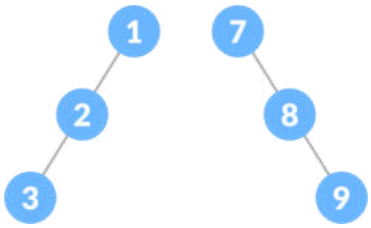


Perfect Binary Tree
A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



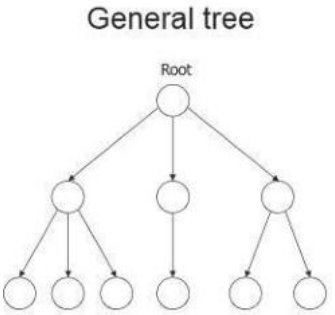
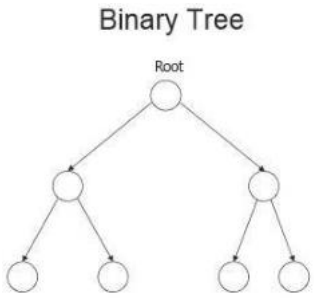


Skewed Binary Tree
A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.



Difference between Tree and Binary Tree

General Tree	Binary Tree
A general tree is a **data structure** in that each node can have infinite number of children,	A Binary tree is a data structure in that each node has at most **two nodes** left and right.
A General tree **can't** be empty.	A Binary tree can be **empty**.
There is no limit on the **degree of node** in a general tree.	Nodes in a binary tree cannot have more than **degree 2**.
Subtree of general tree are **not ordered**.	Subtree of binary tree are **ordered**.
In general tree, root have **in-degree 0** and maximum **out-degree n**.	In binary tree, root have **in-degree 0** and maximum **out-degree 2**.

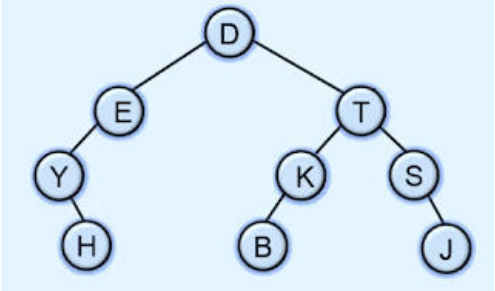
In general tree, each **node** have in-degree **one** and maximum out-degree **n**.	In binary tree, each node have in-degree **one** and maximum out-degree **2**.
Height of a general tree is the length of longest path from root to the leaf of tree. Height(T) = {**max**(height(child1) , height(child2) , … height(child-n)) +1}	Height of a binary tree is : Height(T) = { **max** (Height(Left Child) , Height(Right Child) + 1}
	

Binary Tree Representations

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

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

Consider the following binary tree...



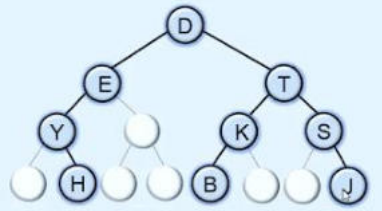
1. Array Representation of Binary Tree

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

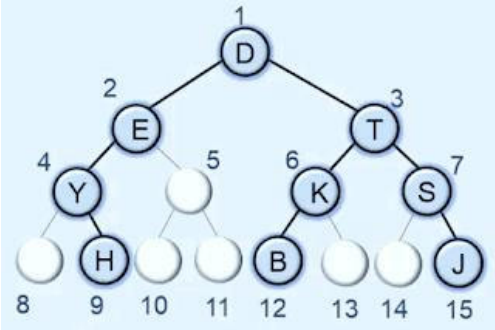
To **represent** a **binary tree** using **array** first we need to convert a binary tree into a full binary tree. and then we give the number to each node and store it into their respective locations.

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

here in the above example to convert this binary tree into a full binary tree we need to add nodes that don't have child nodes till the last level of the tree.



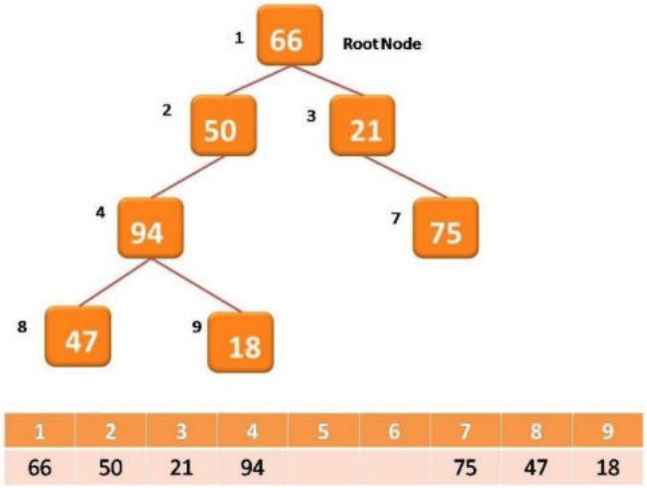
So now the tree becomes a full binary tree. after that to represent it using an array we need to give the numbers to each and every node but level by level.



After giving the number to each and every node now we need to create an array of size 15 + 1.

after that store each one node in array in their respective index points. like D has number 1 then we store it in the array at index 1 and E has number 2 then we store it at index 2 in the array.

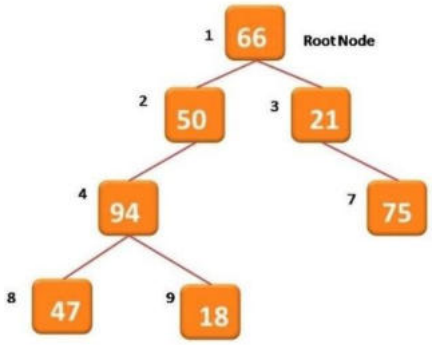
so, this is the array representation of a binary tree.

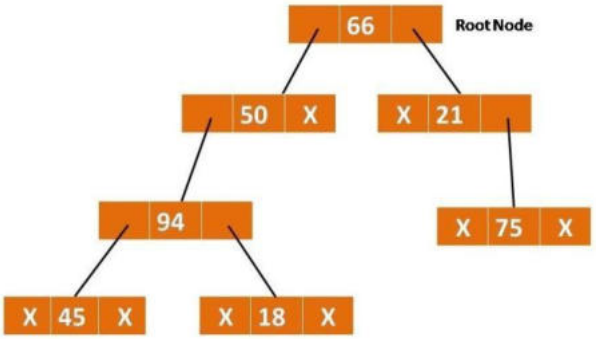


2. Linked List Representation of Binary Tree

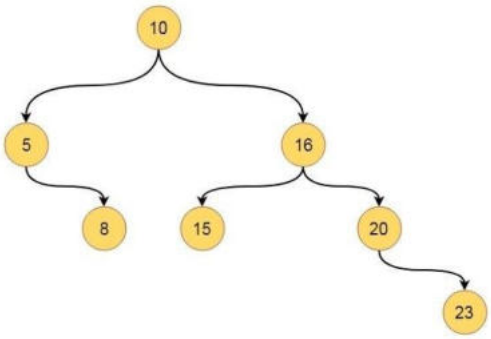
We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

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





Do it for Array and Linked List



Binary Tree Traversals

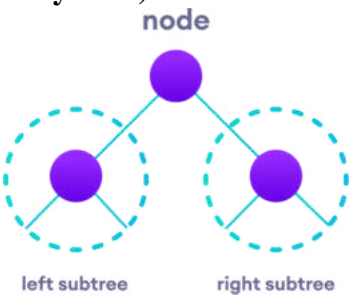
Traversing a tree means visiting every node in the tree.

Tree traversal is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traversal.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are Four types of binary tree traversals.

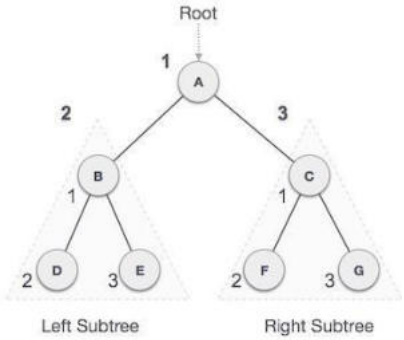
1. **In - Order Traversal (LNR)** (Left subtree -> root -> right subtree)
2. **Pre - Order Traversal** (NLR)
3. **Post - Order Traversal**(LRN)
4. **Level order Traversal(level by level)**



Preorder traversal

Root → Left → Right

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed −

Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

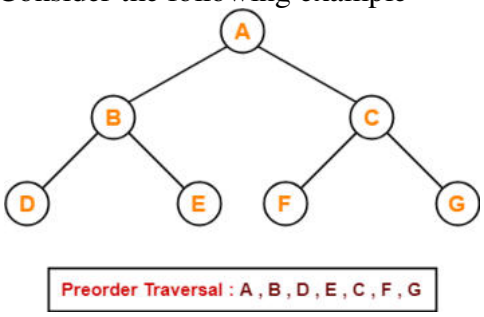
Step 3 − Recursively traverse right subtree.

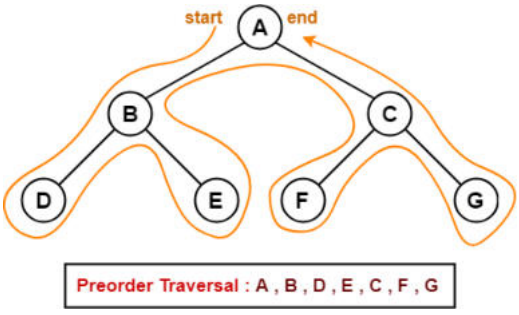
To **traverse a binary tree in preorder**, following operations are carried out:

1. Visit the root.
2. Traverse the left sub tree of root.
3. Traverse the right sub tree of root.

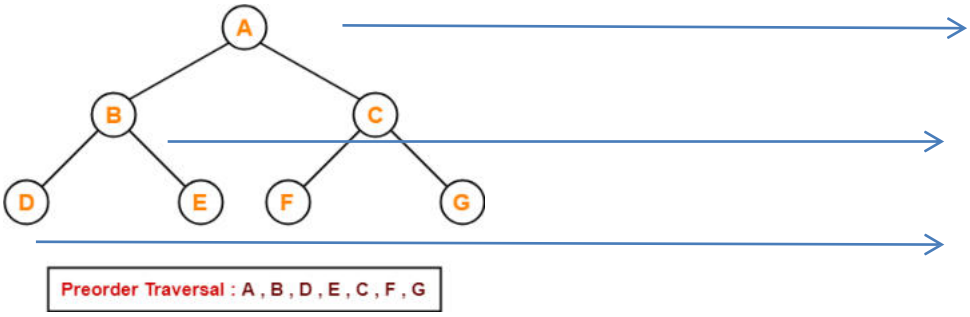
Note: Preorder traversal is also known as NLR traversal. **Root → Left → Right**

Consider the following example-





Level order traversal:



A,B,C,D,E,F,G ====LEVEL ORDER TRAVERSAL….

Inorder traversal (LNR) **(Left subtree -> root -> right subtree)**

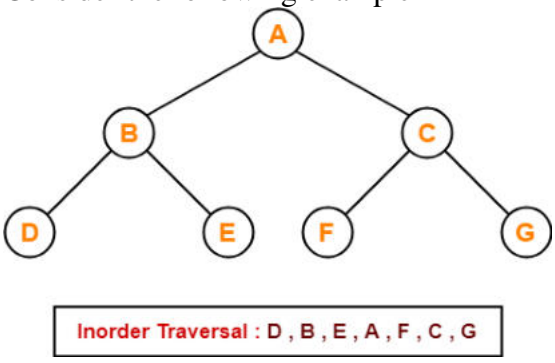
In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

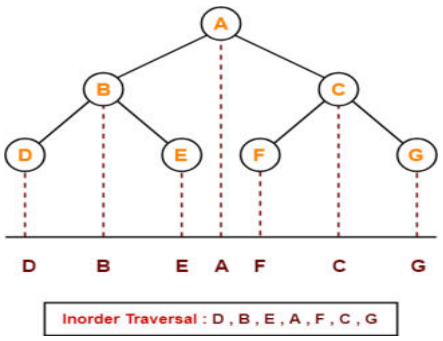
1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

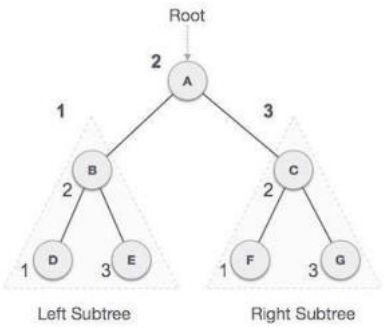
Left → Root → Right

Example-

Consider the following example-







We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed −

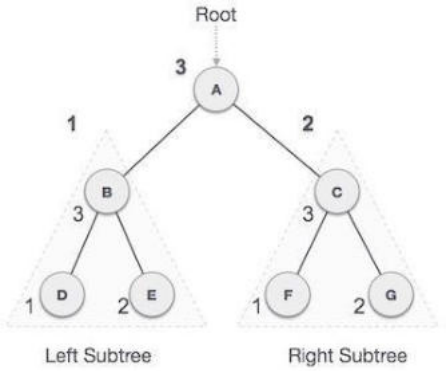
Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Postorder traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

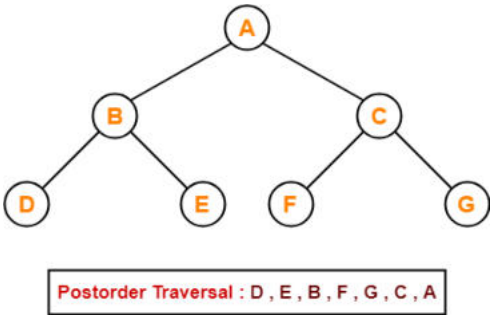
Step 2 − Recursively traverse right subtree.

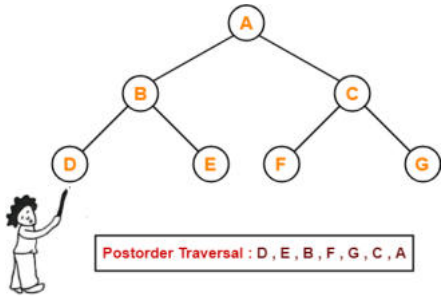
Step 3 − Visit root node.

Left → Right → Root

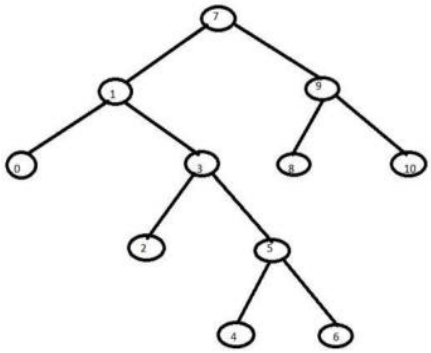
Example-

Consider the following example-





Pluck all the leftmost leaf nodes one by one.



Preorder traversal of the above tree will be: **7,1,0,3,2,5,4,6,9,8,10**
inorder traversal of above tree will be: **0,1,2,3,4,5,6,7,8,9,10**
postorder traversal of the above tree will be: **0,2,4,6,5,3,1,8,10,9,7**

Binary Search Trees:

Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.

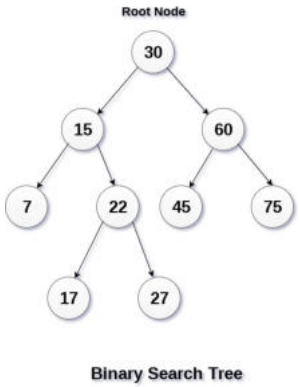
In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.

Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.

This rule will be recursively applied to all the left and right sub-trees of the root.

The properties that separate a binary search tree from a regular binary tree is

- All nodes of left subtree are less than the root node
- All nodes of right subtree are more than the root node
- Both subtrees of each node are also BSTs i.e. they have the above two properties



Binary Search Tree

Advantages of using binary search tree

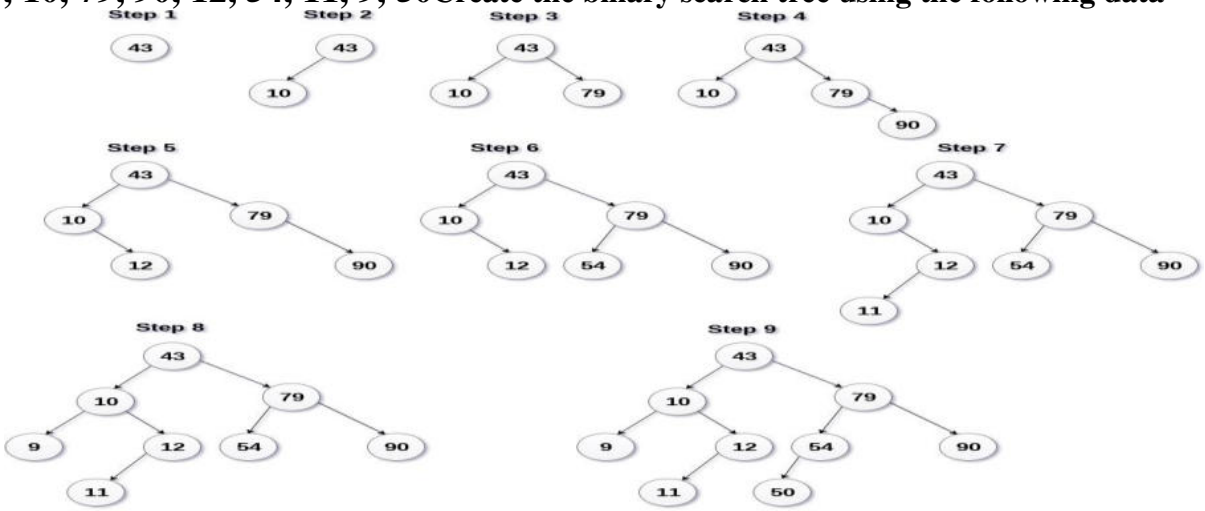
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o(log2n) 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.

Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

- Insert 43 into the tree as the root of the tree.
- Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
- Otherwise, insert it as the root of the right of the right sub-tree.
- The process of creating BST by using the given elements, is shown in the image below.

- **43, 10, 79, 90, 12, 54, 11, 9, 50Create the binary search tree using the following data**



Binary search Tree Creation

elements.

44, 11, 78, 92, 22, 14, 13, 91, 51,10,4

Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

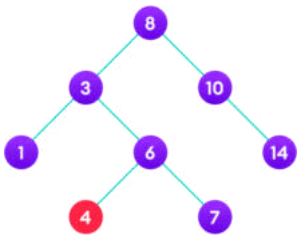
SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

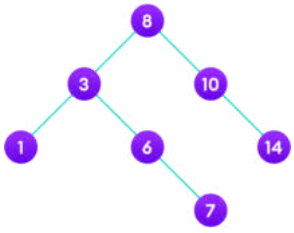
Deletion Operation

There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

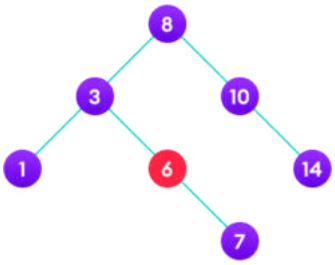


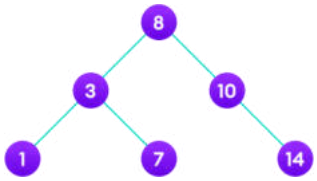


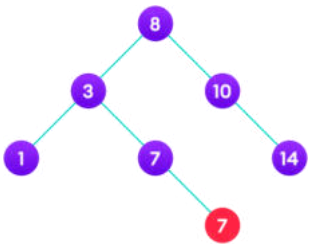
Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.



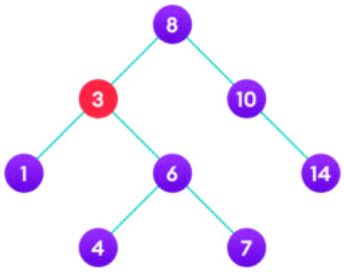


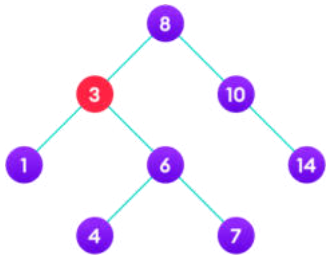


Case III

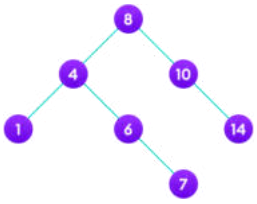
In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.





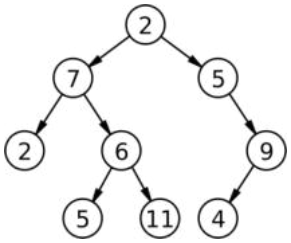
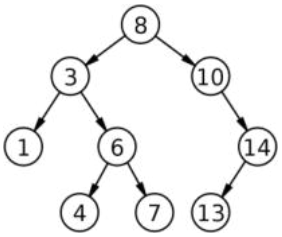
Copy the value of the inorder successor (4) to the node



Home work: Draw the BST for:

Insert the following elements to form a binary search tree: 11, 2, 9, 13, 57, 25, 1, 1, 90, 3. The first number to be inserted is 11.

Binary Tree vs Binary Search Tree –

Basis for Comparison	Binary Tree	Binary Search Tree
Definition	A Binary Tree is a non-linear data structure in which a node can have 0, 1 or 2 nodes. Individually, each node consists of a left pointer, right pointer and data element.	A Binary Search Tree is an organized binary tree with a structured organization of nodes. Each subtree must also be of that particular structure.
Structure	There is no required organization structure of the nodes in the tree.	The values of left subtree of a particular node should be lesser than that node and the right subtree values should be greater.
Operations Performed	The operations that can be performed are deletion, insertion and traversal	As these are sorted binary trees, they are used for fast and efficient binary search, insertion and deletion.
Types	There are several types. Most common ones are the Complete Binary Tree, Full Binary Tree, Extended Binary Tree	The most popular ones are AVL Trees, Splay Trees, Tango Trees, T-Trees.
Example		

Binary Search Tree Applications

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel

AVL TREES:

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.

(or)

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

Why AVL Tree?

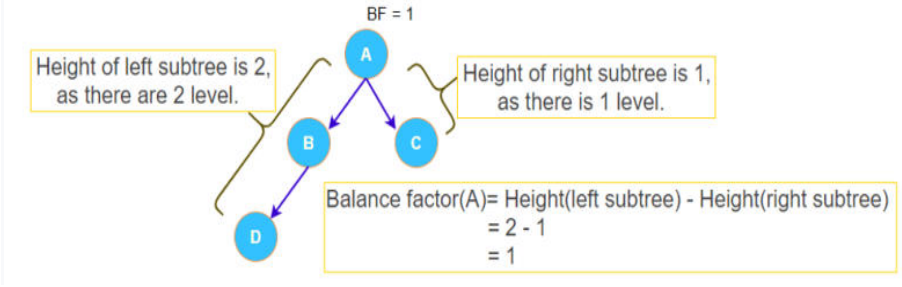
AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes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

Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

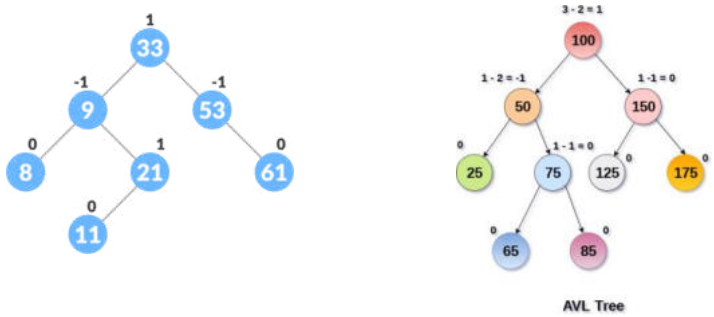
we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree



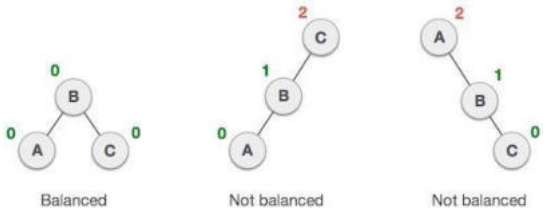
The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

An example of a balanced avl tree is:



AVL Tree

Check the below trees are AVL trees or not?



NOTE: ***Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.***

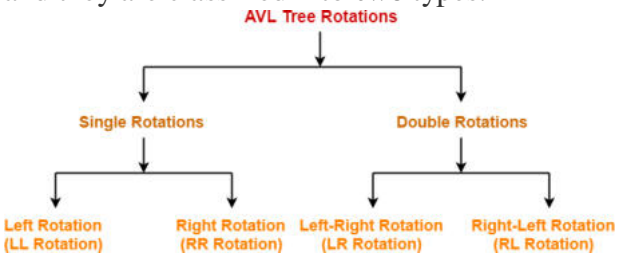
AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

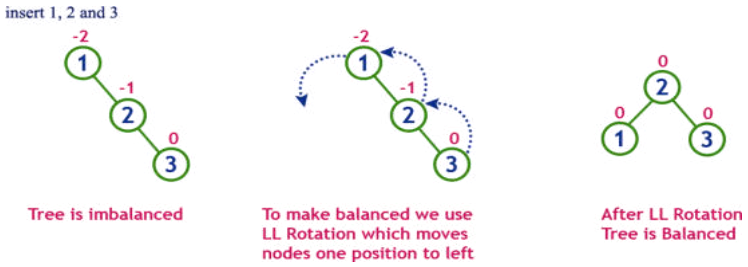
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are **four** rotations and they are classified into **two** types.



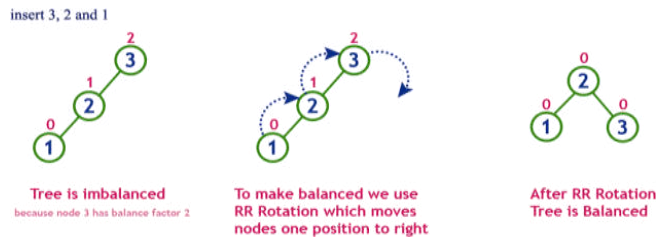
Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



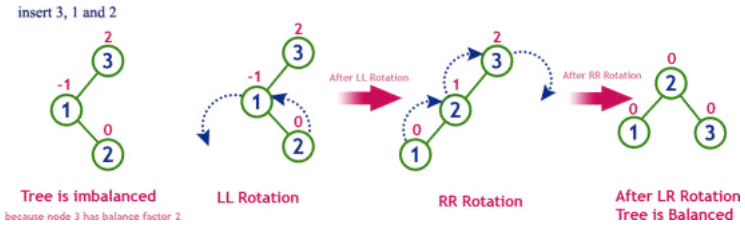
Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



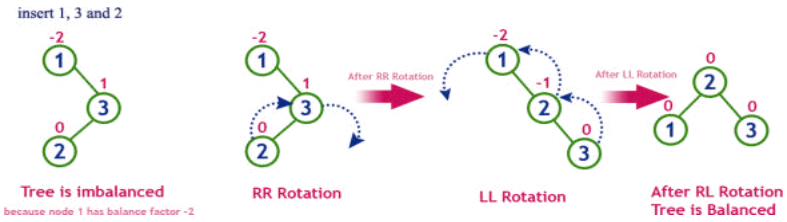
Left Right Rotation (LR Rotation)

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



Right Left Rotation (RL Rotation)

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



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6 - If search element is larger, then continue the search process in right subtree.
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

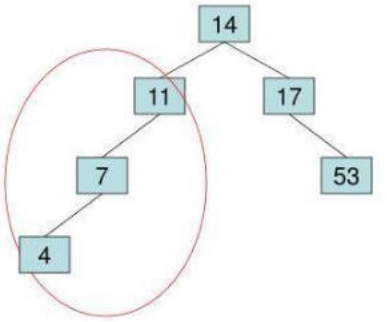
Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

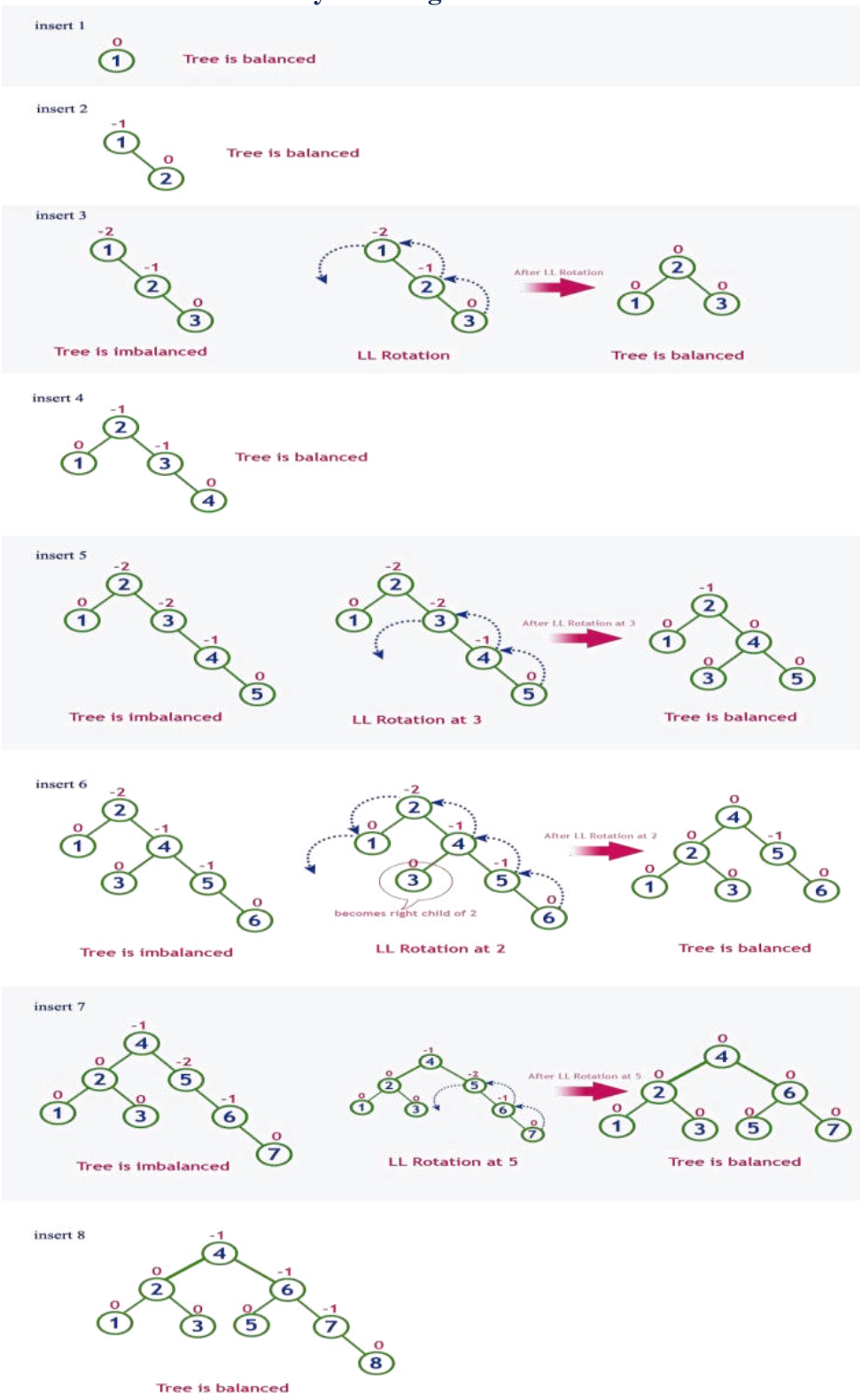
Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
Step 2 - After insertion, check the **Balance Factor** of every node.
Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

AVL Tree Example:

• Insert 14, 17, 11, 7, 53, 4, 13 into an empty A



Example: Construct an AVL Tree by inserting numbers from 1 to 8.



Threaded Binary Tree:

A binary tree can be represented using array representation or linked list representation. When a binary tree is represented using linked list representation, the reference part of the node which doesn't have a child is filled with a NULL pointer. In any binary tree linked list representation, there is a number of NULL pointers than actual pointers.

A. J. Perlis and C. Thornton have proposed new binary tree called "***Threaded Binary Tree***", which makes use of NULL pointers to improve its traversal process. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as ***threads***.

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.

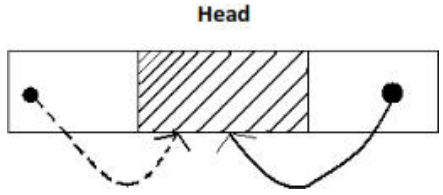
- **Method 1:-** Represent thread a –ve address.
- **Method 2:-** To have a separate Boolean flag for each of left and right pointers, node structure for this is given below,

Alternate node for threaded binary tree.

- LTHREAD = true = Denotes leaf thread link.
- LTHREAD = false = Denotes leaf structural link.
- RTHREAD = true = Denotes right threaded link.
- RTHREAD = false = Denotes right structural link.

Head Node

Head node is simply another node which serves as the predecessor and successor of first and last tree nodes. Tree is attached to the left branch of the head node



Advantages

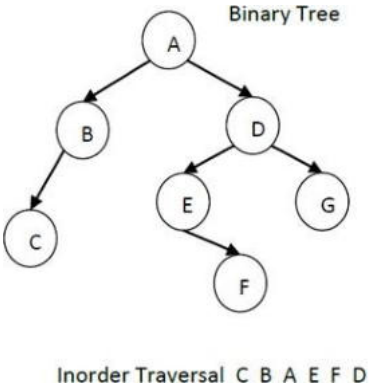
- Inorder traversal is faster than unthreaded version as stack is not required.
- Effectively determines the predecessor and successor for inorder traversal, for unthreaded tree this task is more difficult.
- A stack is required to provide upward pointing information in tree which threading provides.
- It is possible to generate successor or predecessor of any node without having over head of stack with the help of threading.

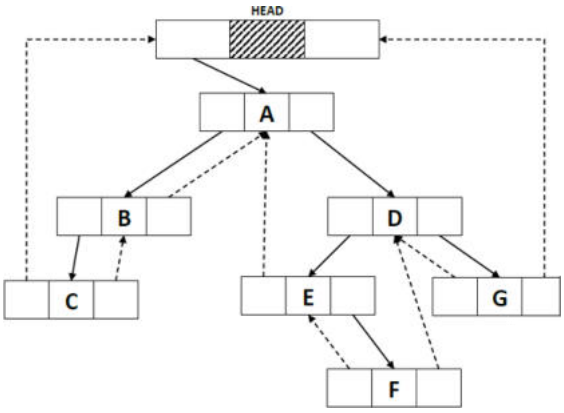
Disadvantages

- Threaded trees are unable to share common subtrees.

- If –ve addressing is not permitted in programming language, two additional fields are required.
- Insertion into and deletion from threaded binary tree are more time consuming because both thread and structural link must be maintained.

Given a Binary Tree

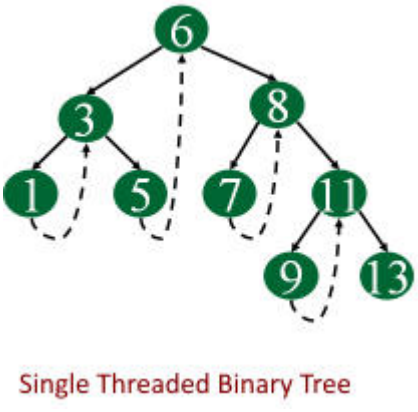


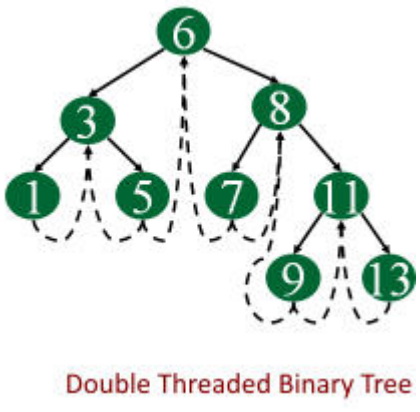


Fully In-threaded binary tree of given binary tree

Types of threaded binary trees:

- **Single Threaded**: each node is threaded towards either the in-order predecessor or successor (left **or** right) means all right null pointers will point to inorder successor **OR** all left null pointers will point to inorder predecessor.
- **Double threaded:** each node is threaded towards both the in-order predecessor and successor (left **and** right) means all right null pointers will point to inorder successor **AND** all left null pointers will point to inorder predecessor.





Hashing:

What is Hashing?

Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.

- Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary

search.

- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

What is Hash Function?

- A fixed process converts a key to a hash key is known as a **Hash Function.**
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

What is Hash Table?

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class.
- Hash table is synchronized and contains only unique elements.

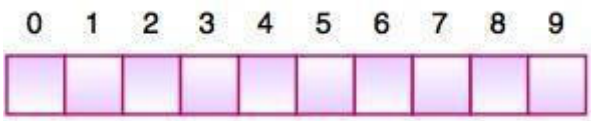


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.
- Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table

- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0

18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

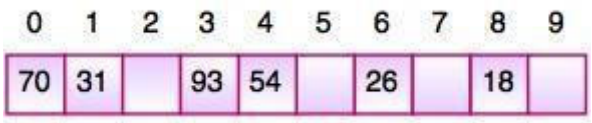


Fig. Hash Table

- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example , λ = 6/10.
- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

Collision in Hashing-

In hashing,

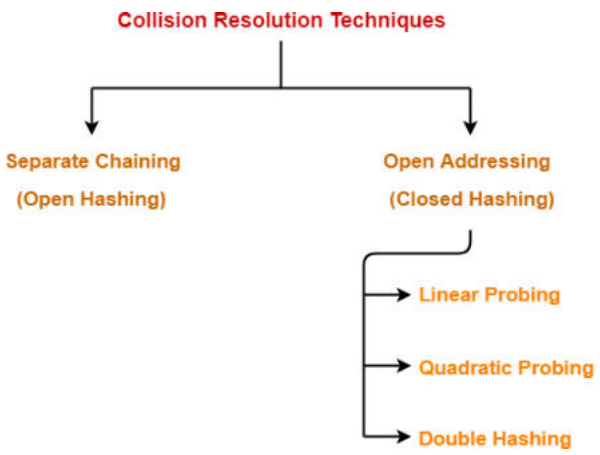
- Hash function is used to compute the hash value for a key.
- Hash value is then used as an index to store the key in the hash table.
- Hash function may return the same hash value for two or more keys.

NOTE: When the hash value of a key maps to an already occupied bucket of the hash table,
it is called as a Collision.

Collision Resolution Techniques-

Collision Resolution Techniques are the techniques used for resolving or handling the collision.

Collision resolution techniques are classified as-



There are two types of collision resolution techniques.

1. Separate chaining (open hashing)
2. Open addressing (closed hashing)

Separate chaining

In this technique, a linked list is created from the slot in which collision has occurred, after which the new key is inserted into the linked list. This linked list of slots looks like a chain, so it is called **separate chaining**. It is used more when we do not know how many keys to insert or delete.

Time complexity

1. Its worst-case complexity for searching is o(n).
2. Its worst-case complexity for deletion is o(n).

Advantages of separate chaining

1. It is easy to implement.
2. The hash table never fills full, so we can add more elements to the chain.
3. It is less sensitive to the function of the hashing.

Disadvantages of separate chaining

1. In this, cache performance of chaining is not good.
2. The memory wastage is too much in this method.
3. It requires more space for element links.

Separate Chaining-

To handle the collision,

- This technique creates a linked list to the slot for which collision occurs.
- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as **separate chaining**.

Problem-

Using the hash function „key mod 7“, insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

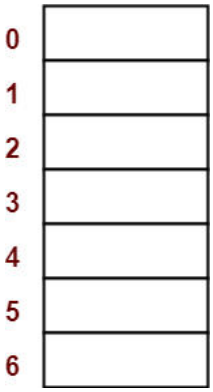
Use separate chaining technique for collision resolution.

Solution-

The given sequence of keys will be inserted in the hash table as-

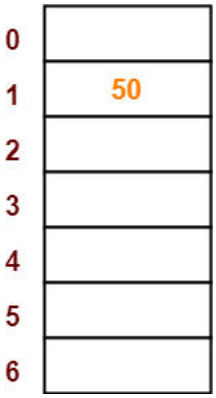
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



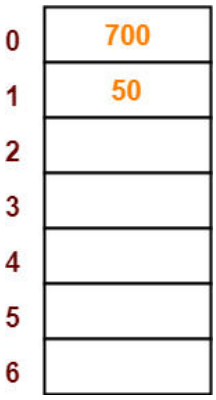
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



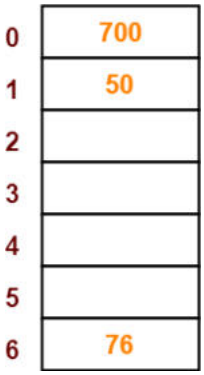
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



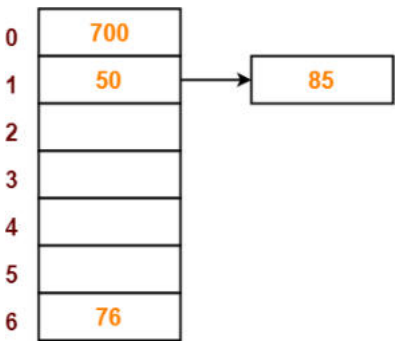
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



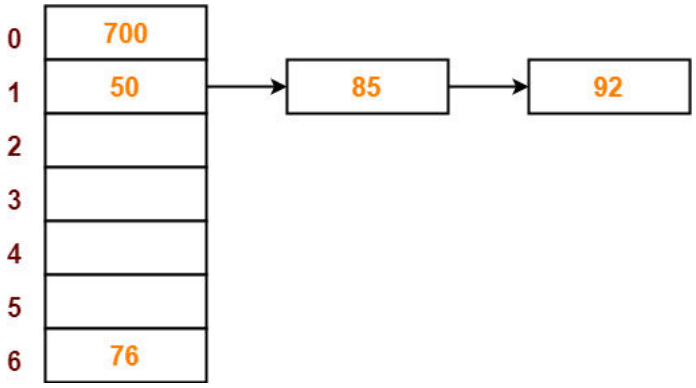
Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



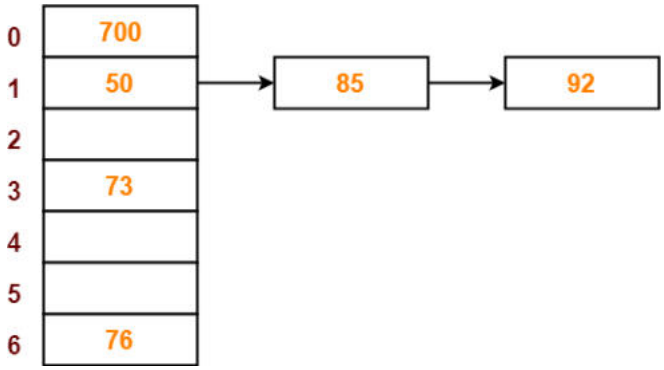
Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-



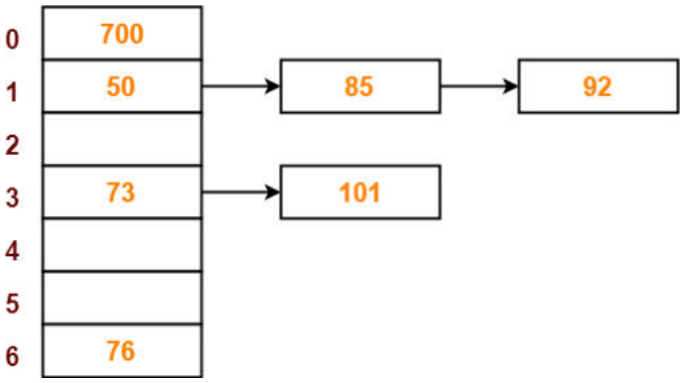
Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- So, key 73 will be inserted in bucket-3 of the hash table as-



Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as-



Open addressing

Open addressing is collision-resolution method that is used to control the collision in the hashing table. There is no key stored outside of the hash table. Therefore, the size of the hash table is always greater than or equal to the number of keys. It is also called **closed hashing**.

The following techniques are used in open addressing:

1. Linear probing
2. Quadratic probing
3. Double hashing

Linear probing

In this, when the collision occurs, we perform a linear probe for the next slot, and this probing is performed until an empty slot is found. In linear probing, the worst time to search for an element is O(table size). The linear probing gives the best performance of the cache but its problem is clustering. The main advantage of this technique is that it can be easily calculated.

Disadvantages of linear probing

1. The main problem is clustering.
2. It takes too much time to find an empty slot.

1. Linear Probing-

In linear probing,

- When collision occurs, we linearly probe for the next bucket.
- We keep probing until an empty bucket is found.

Problem-

Using the hash function „key mod 7“, insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

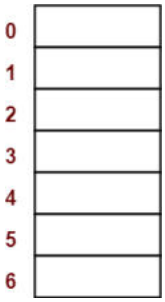
Use linear probing technique for collision resolution.

Solution-

The given sequence of keys will be inserted in the hash table as-

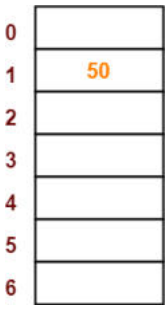
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



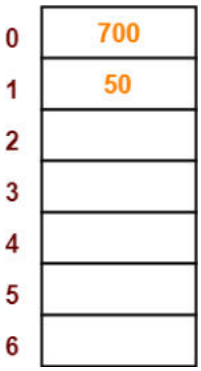
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-

0	700
1	50
2	
3	
4	
5	
6	76

Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-2.
- So, key 85 will be inserted in bucket-2 of the hash table as-

0	700
1	50
2	85
3	
4	
5	
6	76

Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-3.
- So, key 92 will be inserted in bucket-3 of the hash table as-

0	700
1	50
2	85
3	92
4	
5	
6	76

Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-4.
- So, key 73 will be inserted in bucket-4 of the hash table as-

0	700
1	50
2	85
3	92
4	73
5	
6	76

Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-5.
- So, key 101 will be inserted in bucket-5 of the hash table as-

0	700
1	50
2	85
3	92
4	73
5	101
6	76

Quadratic probing
In this, when the collision occurs, we probe for **i^{2th}** slot in **i^{th}** iteration, and this probing is performed until an empty slot is found. The cache performance in quadratic probing is lower than the linear probing. Quadratic probing also reduces the problem of clustering.

In quadratic probing,

- When collision occurs, we probe for i^2,,th bucket in i^{th} iteration.
- We keep probing until an empty bucket is found.

Double hashing
In this, you use another hash function, and probe for (i * hash 2(x)) in the i^{th} iteration. It takes longer to determine two hash functions. The double probing gives the very poor the cache performance, but there has no clustering problem in it.

In double hashing,

- We use another hash function hash2(x) and look for i * hash2(x) bucket in i^{th} iteration.
- It requires more computation time as two hash functions need to be computed.

<u>Comparison of Open Addressing Techniques-</u>

	Linear Probing	Quadratic Probing	Double Hashing
Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Separate Chaining Vs Open Addressing-

Separate Chaining	Open Addressing
Keys are stored inside the hash table as well as outside the hash table.	All the keys are stored only inside the hash table. No key is present outside the hash table.
The number of keys to be stored in the hash table can even exceed the size of the hash table.	The number of keys to be stored in the hash table can never exceed the size of the hash table.
Deletion is easier.	Deletion is difficult.
Extra space is required for the pointers to store the keys outside the hash table.	No extra space is required.
Cache performance is poor. This is because of linked lists which store the keys outside the hash table.	Cache performance is better. This is because here no linked lists are used.
Some buckets of the hash table are never used which leads to wastage of space.	Buckets may be used even if no key maps to those particular buckets.

UNIT - III

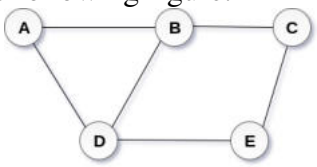
GRAPHS: INTRODUCTION

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

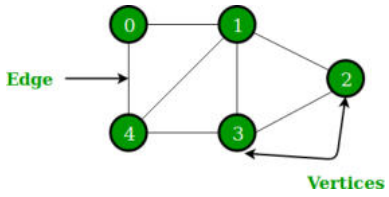
Graph can be defined as:

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Undirected Graph

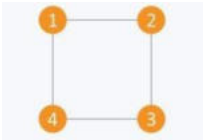


In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

- ✓ Graphs are used to solve many real-life problems.
- ✓ Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.
- ✓ Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node).
- ✓ Each node is a structure and contains information like person id, name, gender, locale etc.

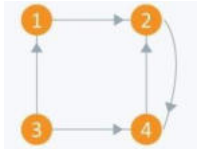
Types of graphs

- Undirected: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.



Undirected Graph

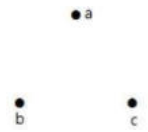
- Directed: A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



Directed Graph

Null Graph(isolated graph)

A **graph having no edges** is called a Null Graph.



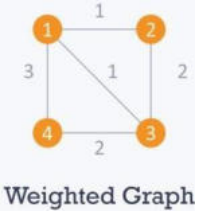
Trivial Graph

A **graph with only one vertex** is called a Trivial Graph.

Example

In the above shown graph, there is only one vertex 'a' with no other edges. Hence it is a Trivial graph.

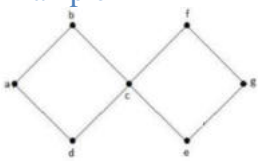
Weighted: In a weighted graph, each edge is assigned a weight or cost.



Cyclic Graph

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

Example

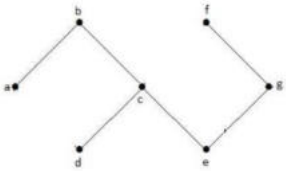


In the above example graph, we have two cycles a-b-c-d-a and c-f-g-e-c. Hence it is called a cyclic graph.

Acyclic Graph

A graph **with no cycles** is called an acyclic graph.

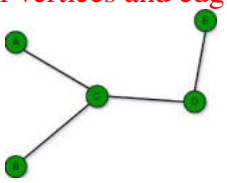
Example



In the above example graph, we do not have any cycles. Hence it is a non-cyclic graph.

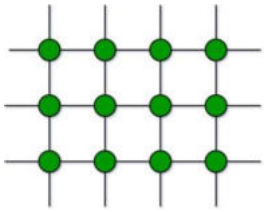
Finite Graph

A graph G= (V, E) in case the number of vertices and edges in the graph is finite in number.

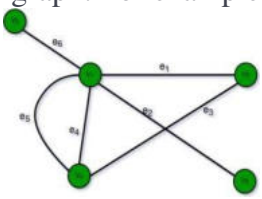


Infinite Graph

A graph G=(V, E) is said to infinite if the number of edges and vertices in the graph is infinite in number.

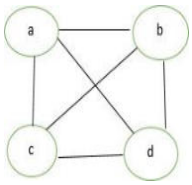


Multi Graph: Any graph which contains some parallel edges but doesn't contain any self-loop is called multi graph. For example A Road Map.

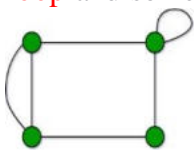


- **Parallel Edges:** If two vertices are connected with more than one edge than such edges are called parallel edges that is many roots but one destination.
- **Loop:** An edge of a graph which join a vertex to itself is called loop or a self-loop.

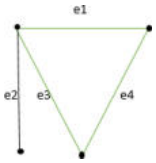
Complete Graph: A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attach with n-1 edges. A complete graph is also called Full Graph.



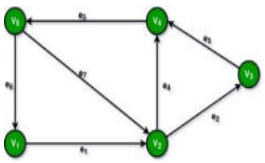
Pseudo Graph: A graph G with a self loop and some multiple edges is called pseudo graph.



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*.



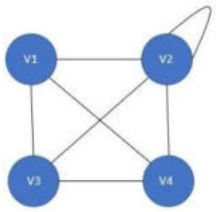
Digraph Graph: A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) is called Digraph. It is also called *Directed Graph*. Ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj.
Here in the figure:
e1 = (V1, V2)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



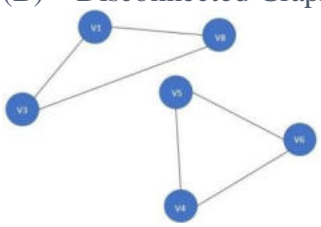
Connected or Disconnected Graph

In case one can find a path from one vertex of the graph to any of the other vertex, then the graph is said to be a connected graph. Thus a null graph is said to a disconnected graph as there is no edge connecting the vertices.

(A) – Connected Graph



(B) – Disconnected Graph



Graph Representations

In graph theory, a graph representation is a technique to store graph into the memory of computer.

To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

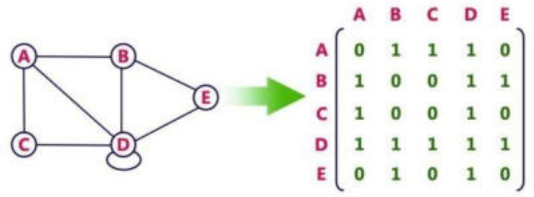
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

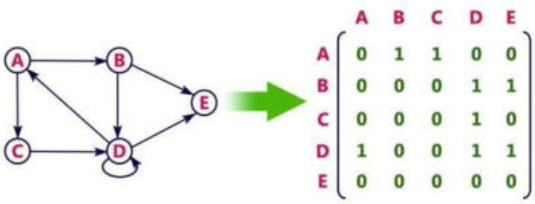
1. Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



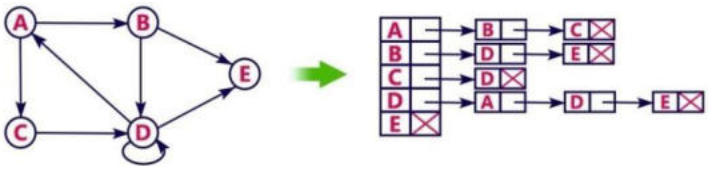
Directed graph representation...



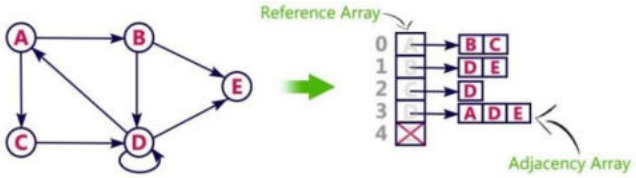
2. Adjacency List

- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a **singly linked list** of neighbors of v.
- N this representation, every vertex of a graph contains list of its adjacent vertices.

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



- This representation can also be implemented using an array as follows..



Pros:

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

Cons:

- The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

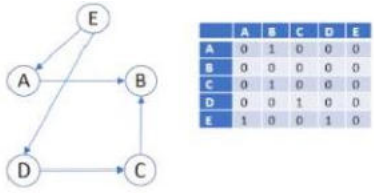


Fig 3: Adjacency Matrix for a directed graph

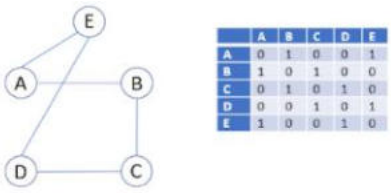


Fig 4: Adjacency Matrix for an undirected graph

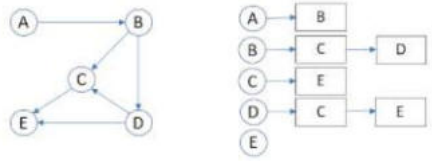


Fig 6: Adjacency list for a directed graph

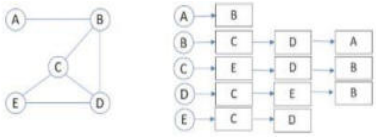


Fig 7: Adjacency list for an undirected graph

Graph traversals

- **Graph traversal** is a technique used to search for a vertex in a graph. It is also used to decide the order of vertices to be visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. This means that, with graph traversal, we can visit all the vertices of the graph without getting into a looping path.
  Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs.

Lets discuss each one of them in detail.

1. Breadth First Search
2. Depth First Search

BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

Procedure for graph traversal using BFS

Step 1: Define a Queue of size total number of vertices in the graph.

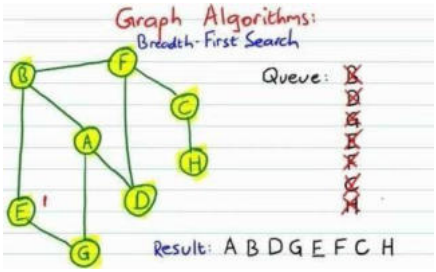
Step 2: Select any vertex as the starting point for traversal.
Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex, that is in front of the Queue and is not visited, and insert them into the Queue.

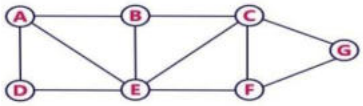
Step 4: When there is no new vertex to visit from the vertex in front of the Queue, delete that vertex from the Queue.

Step 5: Repeat steps 3 and 4 until the queue becomes empty.

Step 6: When the queue becomes Empty, produce the final spanning-tree by removing unused edges from the graph.

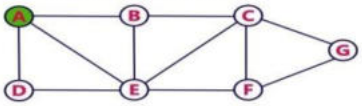


Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

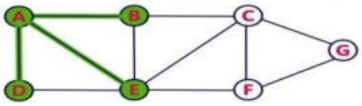


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

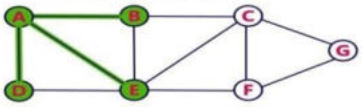


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

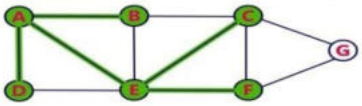


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

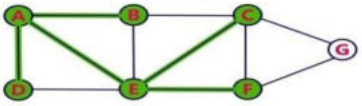


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

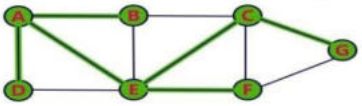


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

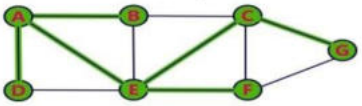


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

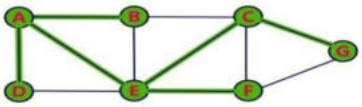


Queue

						G

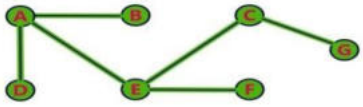
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.

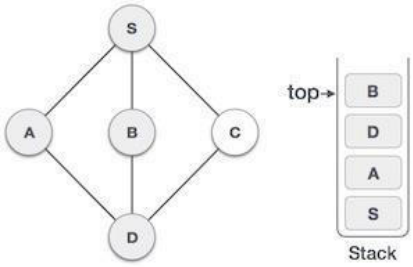


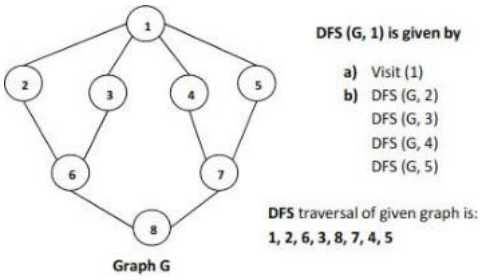
Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



DFS (Depth First Search)



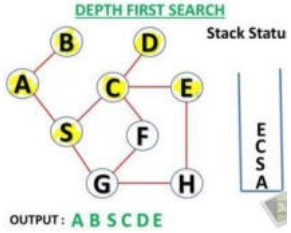


DFS (G, 1) is given by

a) Visit (1)
b) DFS (G, 2)
DFS (G, 3)
DFS (G, 4)
DFS (G, 5)

DFS traversal of given graph is:
1, 2, 6, 3, 8, 7, 4, 5

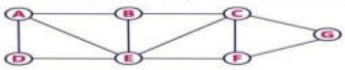
DFS traversal of a graph produces a spanning tree **as final result.** Spanning Tree **is a graph without loops. We use** Stack data structure **with maximum size of total number of vertices in the graph to implement DFS traversal.**



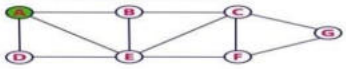
Procedure for graph traversal using DFS

Step 1: Define a Stack of size total number of vertices in the graph.
Step 2: Select any vertex as the starting point for traversal.
Visit that vertex and push it on to the Stack.
Step 3: Visit any one of the adjacent vertices of the vertex, that is at top of the stack and is not visited, and push it on to the stack.
Step 4: Repeat step 3 until there are no new vertices to visit from each vertex on top of the stack.
Step 5: When there is no new vertex to visit, use backtracking and pop one vertex from the stack.
Step 6: Repeat steps 3, 4, and 5 until stack becomes Empty.
Step 7: When stack becomes Empty, produce the final spanning-tree by removing unused edges from the graph.

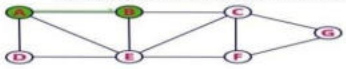
Consider the following example graph to perform DFS traversal



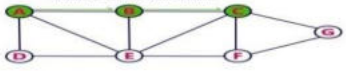
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

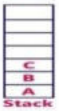


Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

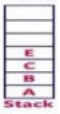


Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



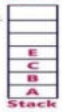


Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



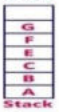
Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

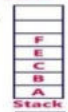


Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

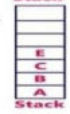
Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



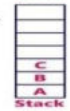
Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



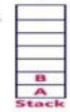
Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

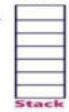


Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

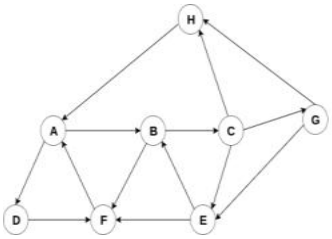


Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Difference between BFS and DFS Binary Tree

BFS	DFS
BFS finds the shortest path to the destination.	DFS goes to the bottom of a subtree, then backtracks.
The full form of BFS is Breadth-First Search	The full form of DFS is Depth First Search.
It uses a queue to keep track of the next location to visit.	It uses a stack to keep track of the next location to visit
BFS traverses according to tree level.	DFS traverses according to tree depth.
It is implemented using FIFO list.	It is implemented using LIFO list.
It requires more memory as compare to DFS.	It requires less memory as compare to BFS.
This algorithm gives the shallowest path solution.	This algorithm doesn't guarantee the shallowest path solution.
There is no need of backtracking in BFS.	There is a need of backtracking in DFS.
You can never be trapped into finite loops.	You can be trapped into infinite loops.
If you do not find any goal, you may need to expand many nodes before the solution is found.	If you do not find any goal, the leaf node backtracking may occur.

Applications of BFS

Here, are Applications of BFS:

Un-weighted Graphs:
BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of the graph in the shortest time possible with high accuracy.

P2P Networks:
BFS can be implemented to locate all the nearest or neighboring nodes in a peer to peer network. This will find the required data faster.

Web Crawlers:
Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all the links from that source.

Network Broadcasting:
A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

Applications of DFS
Here are Important applications of DFS:

Weighted Graph:
In a weighted graph, DFS graph traversal generates the shortest path tree and minimum spanning tree.

Detecting a Cycle in a Graph:
A graph has a cycle if we found a back edge during DFS. Therefore, we should run DFS for the graph and verify for back edges.

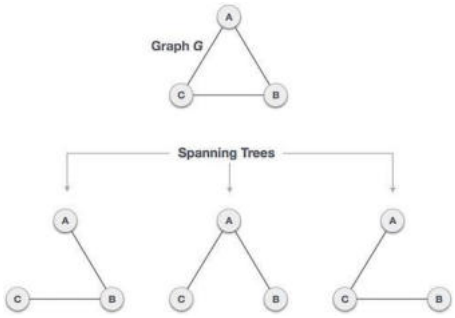
Path Finding:
We can specialize in the DFS algorithm to search a path between two vertices.

Topological Sorting:
It is primarily used for scheduling jobs from the given dependencies among the group of jobs. In computer science, it is used in instruction scheduling, data serialization, logic synthesis, determining the order of compilation tasks.

Searching Strongly Connected Components of a Graph:
It used in DFS graph when there is a path from each and every vertex in the graph to other remaining vertexes.

Solving Puzzles with Only One Solution:
DFS algorithm can be easily adapted to search all solutions to a maze by including nodes on the existing path in the visited set.

Spanning trees
A subset of the graph, in which the vertices are covered with minimum possible number of edges is known as a Spanning tree. Spanning tree cannot be disconnected as it does not have cycles.
It is understood that every connected and undirected graph G has a minimum of one spanning tree and the disconnected graph does not have any spanning tree.

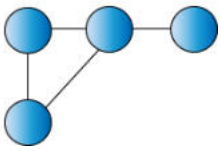


For one complete graph, we identified three spanning trees. The complete undirected graph have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, $\mathbf{3^{3-2}= 3}$ spanning trees are possible.

Spanning Tree:

Given a connected undirected graph, a spanning tree of that graph is a subgraph that is a tree and joined all vertices. A single graph can have many spanning trees.

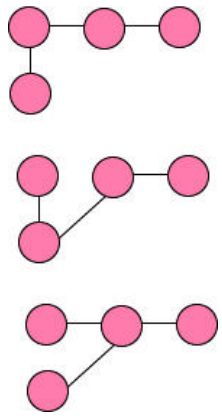
Connected Undirected Graph



IN THIS 4 VERTICES=====4^{4-2} =4^2====4X4=16 ST

For the above-connected graph. There can be multiple spanning Trees like

Spanning Trees



The properties of a spanning tree connected to graph G are -

1. A connected graph G can have more than one spanning tree.
2. All possible spanning trees of graph G, have the same number of edges and vertices.
3. The spanning tree does not have any cycle (loops).
4. Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
5. Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

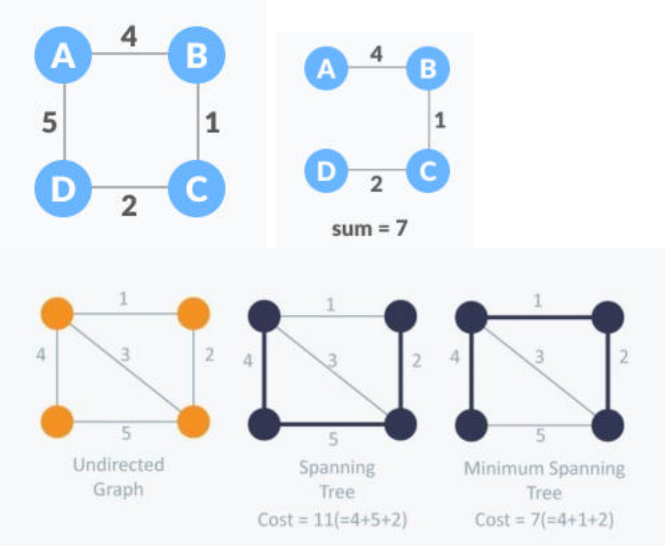
Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Minimum Spanning Tree

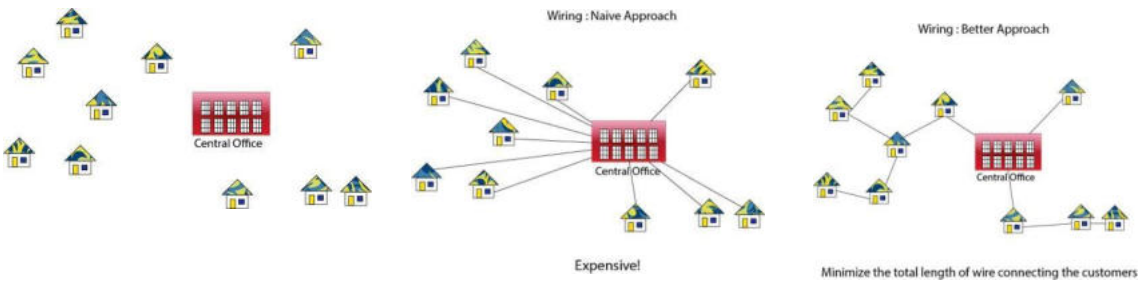
- A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

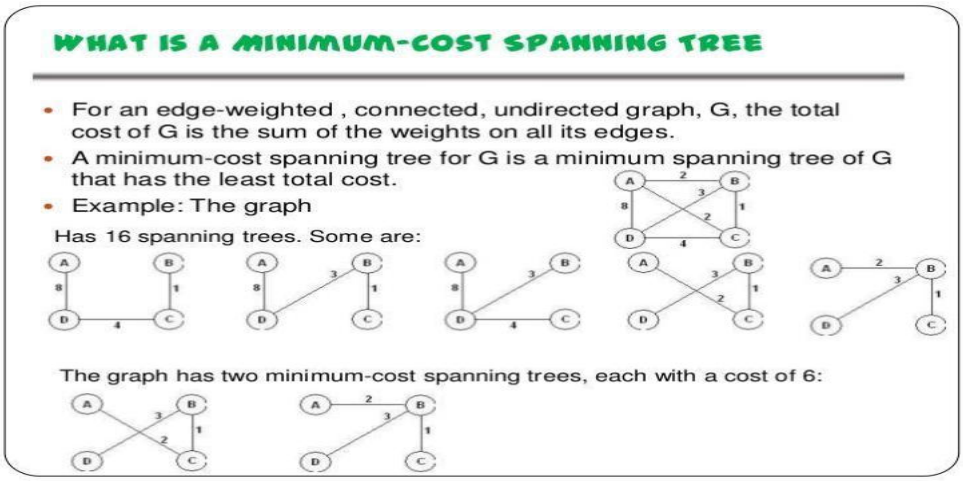


Application of Minimum Spanning Tree

1. Consider n stations are to be linked using a communication network & laying of communication links between any two stations involves a cost.
   The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.
2. Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
3. Designing Local Area Networks.
4. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.
5. Suppose you want to apply a set of houses with
   - Electric Power
   - Water
   - Telephone lines
   - Sewage lines

   To reduce cost, you can connect houses with minimum cost spanning trees.
   For Example, Problem laying Telephone Wire.





There are two Minimum Spanning Tree Algorithms:-

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

The algorithm is given as follows.

Algorithm

Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
Step 4: Add the selected edge and the vertex to the minimum spanning tree T
[END OF LOOP]
Step 5: EXIT

Example :

Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.

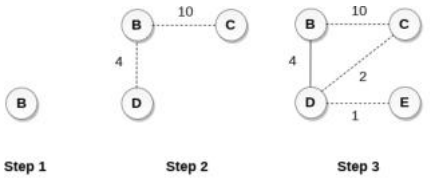
Step 1 : Choose a starting vertex B.
Step 2: Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.
Step 3: Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
Step 3: Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.

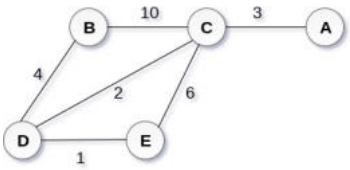
Step 4: Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

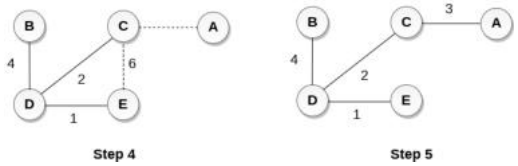
The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

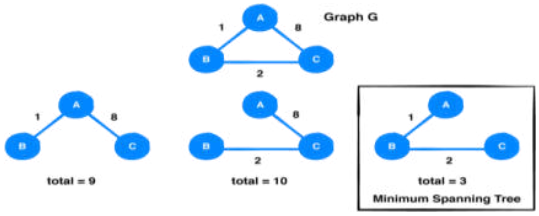
The cost of MST will be calculated as;

cost(MST) = 4 + 2 + 1 + 3 = 10 units.

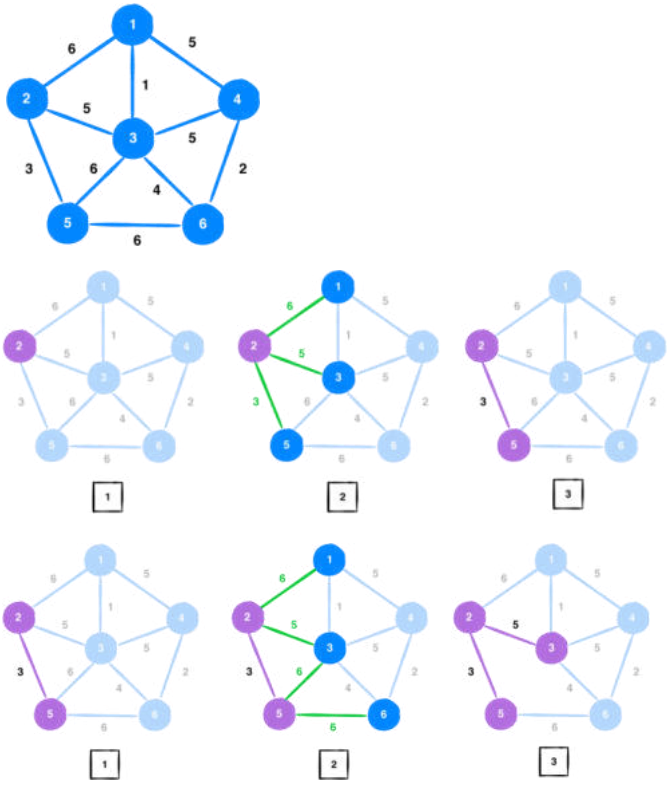


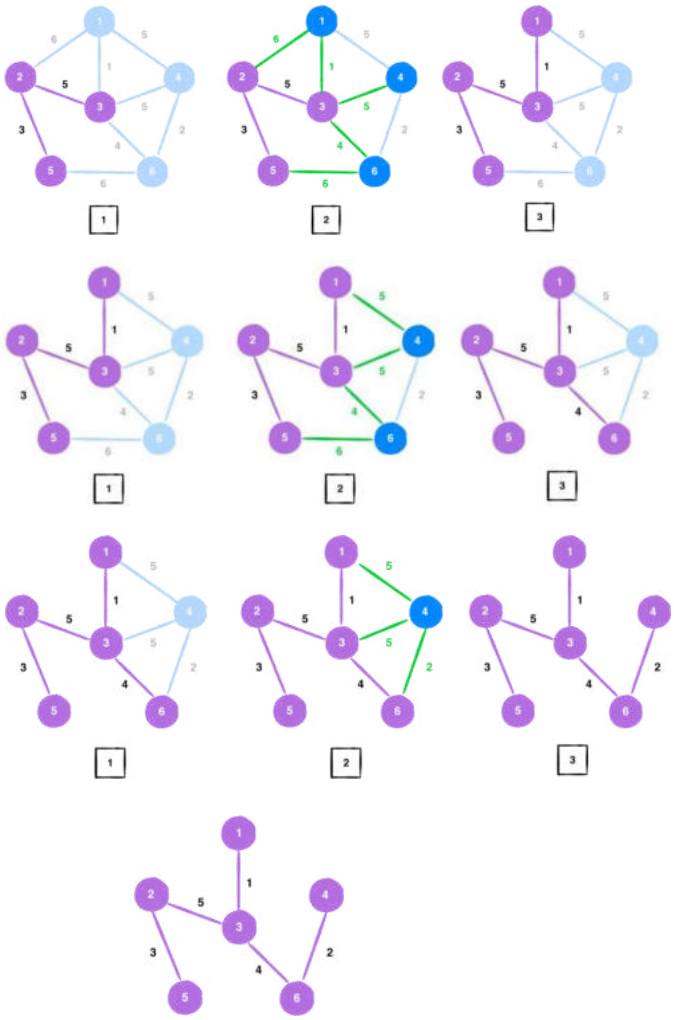






EXAMPLE:





Kruskal's Algorithm

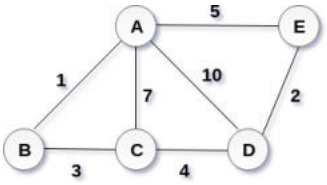
Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

The Kruskal's algorithm is given as follows.

Algorithm

Step 1: Create a forest in such a way that each graph is a separate tree.
Step 2: Create a priority queue Q that contains all the edges of the graph.
Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY
Step 4: Remove an edge from Q
Step 5: IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).
ELSE
Discard the edge
Step 6: END

Example :
Apply the Kruskal's algorithm on the graph given as follows.



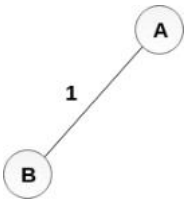
Solution:
The weight of the edges given as :

Edge	**AE**	**AD**	**AC**	**AB**	**BC**	**CD**	**DE**
Weight	5	10	7	1	3	4	2

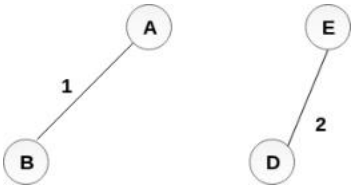
Sort the edges according to their weights.

Edge	**AB**	**DE**	**BC**	**CD**	**AE**	**AC**	**AD**
Weight	1	2	3	4	5	7	10

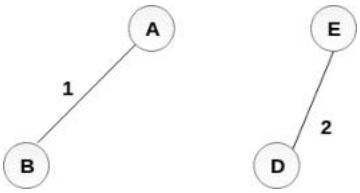
Start constructing the tree;
Add AB to the MST;

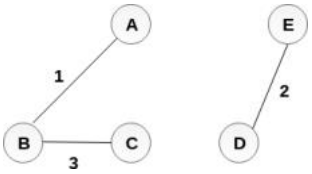


Add DE to the MST;



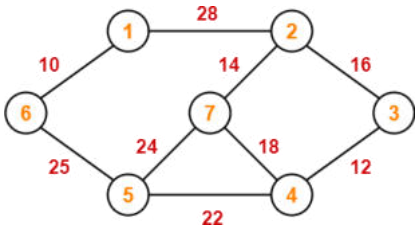
Add BC to the MST;





The next step is to add AE, but we can't add that as it will cause a cycle.
The next edge to be added is AC, but it can't be added as it will cause a cycle.
The next edge to be added is AD, but it can't be added as it will contain a cycle.
Hence, the final MST is the one which is shown in the step 4.
the cost of MST = 1 + 2 + 3 + 4 = 10.

Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm-

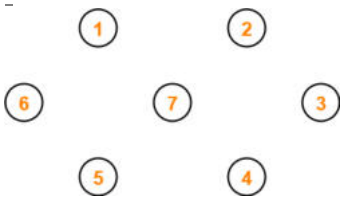


Solution-

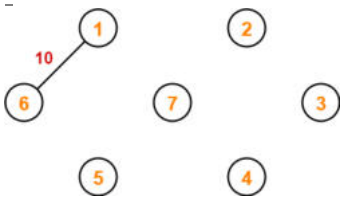
To construct MST using Kruskal's Algorithm,

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

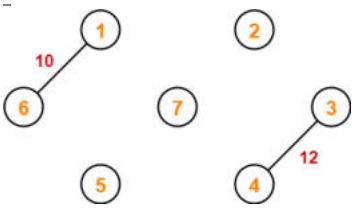
Step-01:



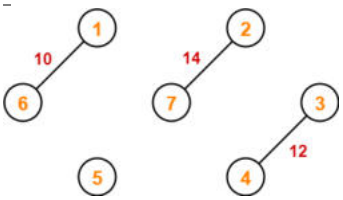
Step-02:



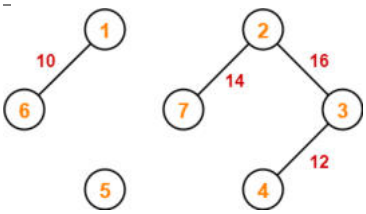
Step-03:



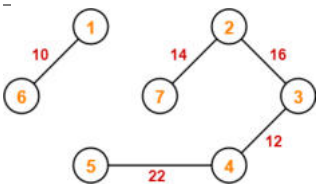
Step-04:



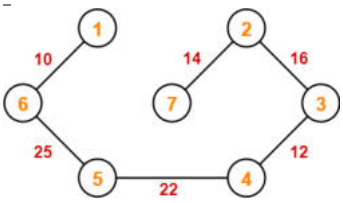
Step-05:



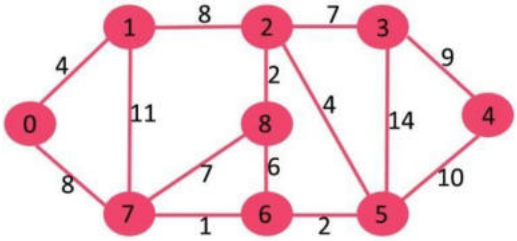
Step-06:



Step-07:



Since all the vertices have been connected / included in the MST, so we stop.
Weight of the MST
= Sum of all edge weights
= 10 + 25 + 22 + 12 + 16 + 14
= 99 units



Draw the Minimum Spanning Tree using Prims Algorithm and Kruskals Algorithm.

Dijkstra's Shortest Path Algorithm

- Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph.

  Dijkstra's algorithm, published in 1959, is named after its discoverer Edsger Dijkstra, who was a Dutch computer scientist. This algorithm aims to find the shortest-path in a directed or undirected graph with non-negative edge weights.

  Problem Statement
  Given a weighted graph G, the objective is to find the shortest path from a given source vertex to all other vertices of G.

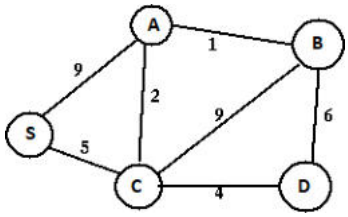
  The graph has the following characteristics-

  1. Set of vertices V
  2. Set of weighted edges E such that (q,r) denotes an **edge** between **vertices** q and r and cost(q,r) denotes its weight
  3. Dijkstra's Algorithm:
  4. This is a single-source shortest path algorithm and aims to find solution to the given problem statement
  5. This algorithm works for both directed and undirected graphs
  6. It works only for connected graphs
  7. The graph should not contain negative edge weights
  8. The algorithm predominantly follows Greedy approach for finding locally optimal solution. But, it also uses Dynamic Programming approach for building globally optimal solution, since the previous solutions are stored and further added to get final distances from the source vertex
  9. The main logic of this algorithm is basedon the following formula- dist[r]=min(dist[r], dist[q]+cost[q][r])
  10. Algorithm-

  Input Data-
- Cost Adjacency Matrix for Graph G, say cost
- Source vertex, say s

  Output Data-
- Spanning tree having shortest path from s to all other vertices in G

Let's try and understand the working of this algorithm using the following example-



Input Graph (Weighted and Connected)

Given the above weighted and connected graph and source vertex s, following steps are used for finding the tree representing shortest path between s and all other vertices-

Step A- Initialize the distance array (dist) using the following steps of algorithm –

- **Step 1**- Set dist[s]=0, S=ϕ // u is the source vertex and S is a 1-D array having all the visited vertices
- **Step 2**- For all nodes v except s, set dist[v]= ∞

Set of visited vertices (S)	S	A	B	C	D
	0	∞	∞	∞	∞

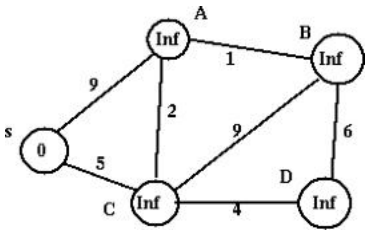


Fig 2: Graph after initializing dist[]

Thus dist[] gets updated as follows-

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞

This updates dist[] as follows-

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞
[s,C]	0	7	14	5	9

Continuing on similar lines, Step B gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞
[s,C]	0	7	14	5	9
[s, C, A]	0	7	8	5	9
[s, C, A, B]	0	7	8	5	9
[s, C, A, B, D]	0	7	8	5	9

The resultant shortest path spanning tree for the given graph is as follows-

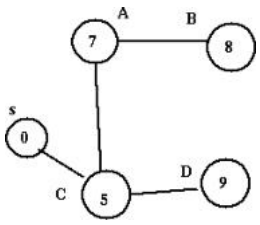


Fig 3: Shortest path spanning tree

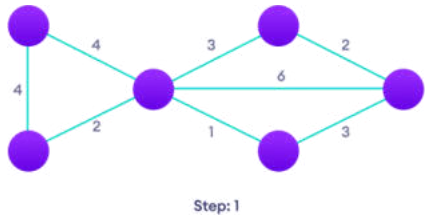
How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

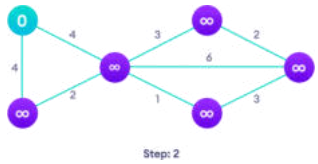
Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.



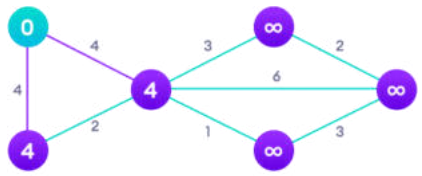
Step: 1

Start with a weighted graph



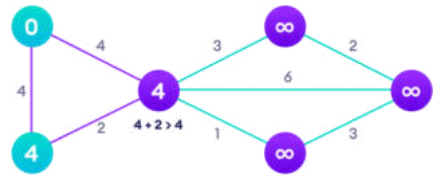
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



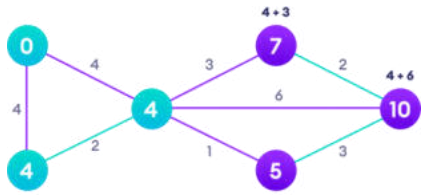
Step: 3

Go to each vertex and update its path length

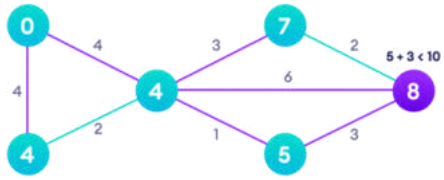


Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it

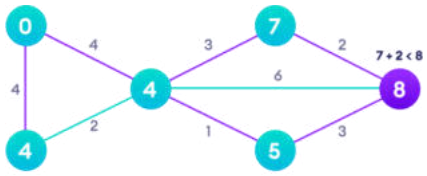


Step: 5



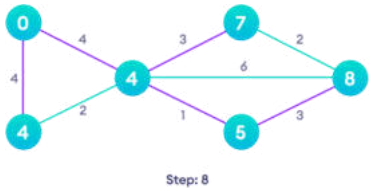
Step: 6

Avoid updating path lengths of already visited vertices
After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs.

Advantages-

Floyd Warshall Algorithm has the following main advantages-

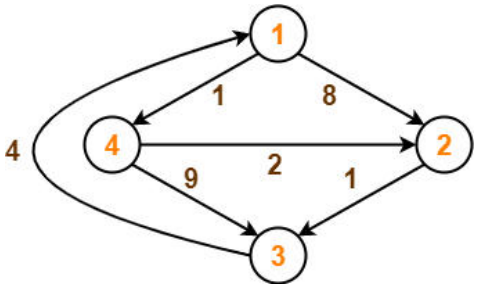
- It is extremely simple.
- It is easy to implement.

When Floyd Warshall Algorithm Is Used?

- Floyd Warshall Algorithm is best suited for dense graphs.
- This is because its complexity depends only on the number of vertices in the given graph.
- For sparse graphs, Johnson's Algorithm is more suitable.

Problem-

Consider the following directed weighted graph-



Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.

Solution-

Step-01:

Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
In the given graph, there are neither self edges nor parallel edges.

Step-02:

- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

Step-03:

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

UNIT - IV

➢ Fundamentals of the Analysis of Algorithm

Algorithm Analysis:

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below −

- ***A priori*** **analysis** − This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.
- ***A posterior*** **analysis** − This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor** − The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor** − The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

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

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.

- Algorithm: SUM(A, B)
- Step 1 - START
- Step 2 - C ← A + B + 10
- Step 3 - Stop

Time Complexity

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

➢ Asymptotic Notation:

- Asymptotic analysis of an algorithm, refers to defining the mathematical foundation/framing of its run-time performance.
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, running time of one operation is

computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases. Similarly the running time of both operations will be nearly same if n is significantly small.

An algorithm falls under three types

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

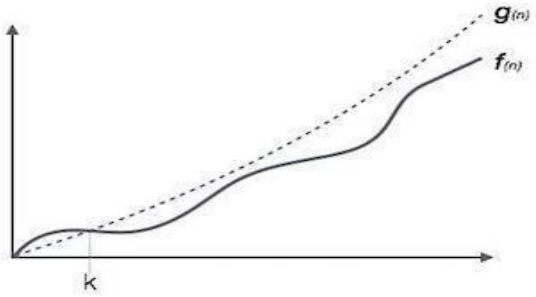
Types of Asymptotic Notations:

Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big-Oh Notation, O

The O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

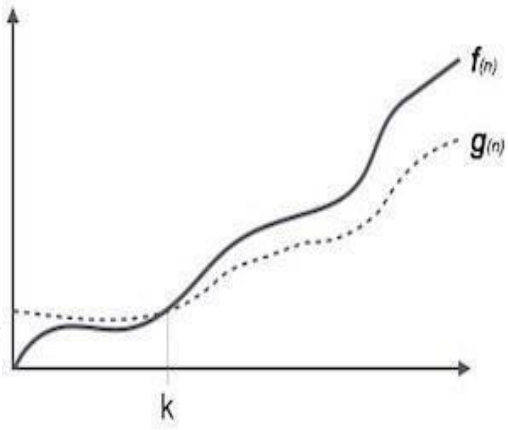


For example, for a function $f(n)$

$O(f(n)) = \{ g(n)$: there exists $c > 0$ and n_0 such that $g(n) \leq c.f(n)$ for all $n > n_0$. $\}$

Big-Omega Notation, Ω

The Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.

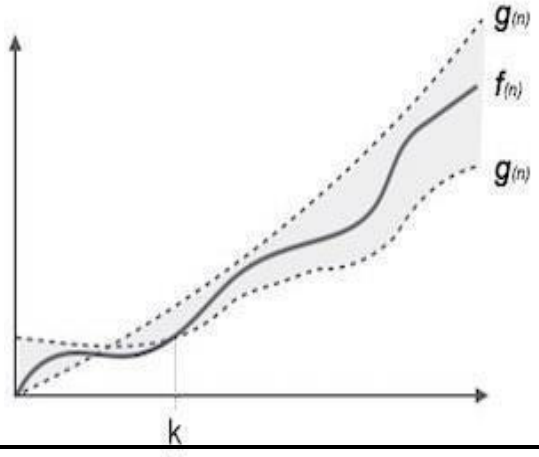


For example, for a function $f(n)$

$\Omega(f(n)) \geq \{ g(n)$: there exists $c > 0$ and n_0 such that $g(n) \leq c.f(n)$ for all $n > n_0$. $\}$

Theta Notation, θ

The θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time.It is represented as following –



$\theta(f(n)) = \{ g(n)$ if $\theta(f(n)) = \{ g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0$. $\}$

Divide and conquer:

It is a one of the process or design technique for solving big problems which requires large amount of input and produce heavy output.

- ***GENERAL METHOD:***

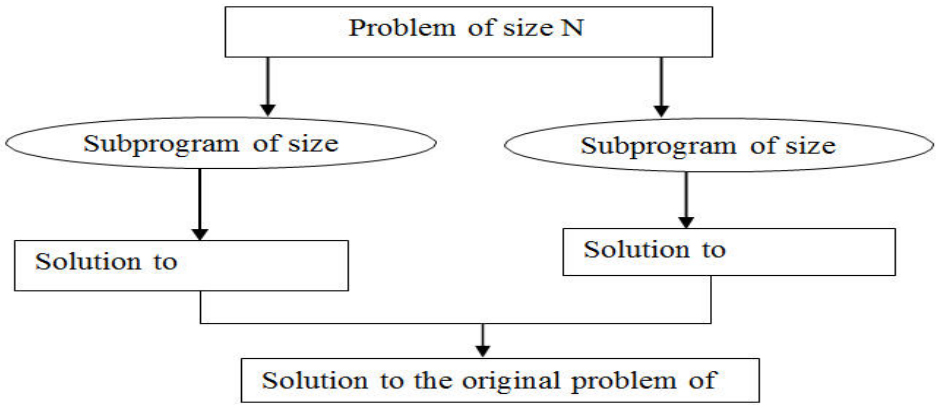
How the Divide and conquer rule work:

In Divide and conquer rule

1st: A problem's instances are divided into several smaller instances of the same problem,in same size.

2nd: And then small instances are solved.

3rd: Then after, the solutions obtained for the smaller instances are combined to get asolution to the original problem.



Diagrammatic representation of Divide and conquer rule

Pseudo code Representation of Divide and conquer rule for problem "P"

```
Algorithm DAndC(P)
{
if small(P) then
return S(P)
else
{
divide P into smaller instances P1,P2,P3…Pk;
apply DAndC to each of these subprograms;  // means DAndC(P1), DAndC(P2)….. DAndC(Pk)
return combine(DAndC(P1), DAndC(P2)….. DAndC(Pk));
}
}
```

P→Problem

Here small(P)→ Boolean value function. If it is true, then the function S is invoked

Calculate time for D and C:

If the problem P of size is n and K sub problems size is n1,n2,n3 ----nk then

$T(n) = g(n)$

$T(n1)+T(n2)+----+T(nk)+f(n)$

T(n)→ Time for DAndC of any Input size n;

g(n)→ Time to compute the answer directly for

small inputsT(n1)→ Time for DAndC of small
input size n1;
T(n2)→ Time for DAndC of small input size n2;
*
*
T(nk)→ Time for DAndC of small input size nk;
F(n)→ Time for dividing P and combining the solution of subproblems.

Time Complexity of DAndC algorithm:

T(n) = T(1) if n=1

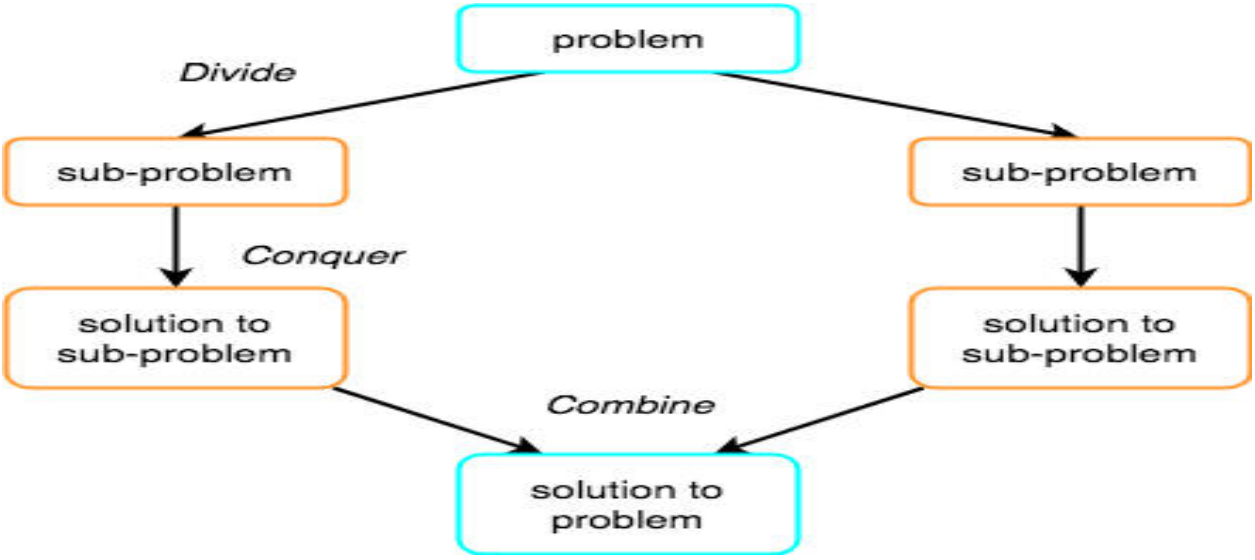
T(n) = aT(n/b)+f(n) if n>1

a,b→ contants.

This is called the **general divide and-conquer recurrence.**

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.
2. **Conquer** the subproblems by solving them. The idea is to break down theproblem into atomic subproblems, where they are actually solved.
3. **Combine** the solutions of the subproblems to find the solution of the actualproblem.



Example for GENERAL METHOD:

As an example, let us consider the problem of computing the sum of n numbers $a_0, \ldots a_{n-1}$. If n > 1, we can divide the problem into two instances of the same problem. They are sum
ofthe first | n/2|numbers Compute the sum of the 1st [n/2] numbers, and then compute the sum of another n/2 numbers.Combine the answers of two n/2 numbers sum.

i.e.,

$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{n/2}) + (a_{n/2} + \ldots\ldots + a_{n-1})$

Assuming that size n is a power of b, to simplify our analysis, we get the followingrecurrence for the running time T(n).

T(n)=aT(n/b)+f(n)

This is called the general **divide and-conquer recurrence**.

f(n)→ is a function that accounts for the time spent on dividing the problem into smaller onesand on combining their solutions. (For the summation example, a = b = 2 and f (n) = 1.

Advantages of D and C:

The time spent on executing the problem using DAndC is smaller than other method.This technique is ideally suited for parallel computation. This approach provides an efficient algorithm in computer science.

Applications of Divide and conquer rule or algorithm:

- Binary search,
- Quick sort,
-
- Merge sort,
- Strassen's matrix multiplication.

➢ Merge sort

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A ofn number of elements. The algorithm processes the elements in 3 steps.

1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of thearray.

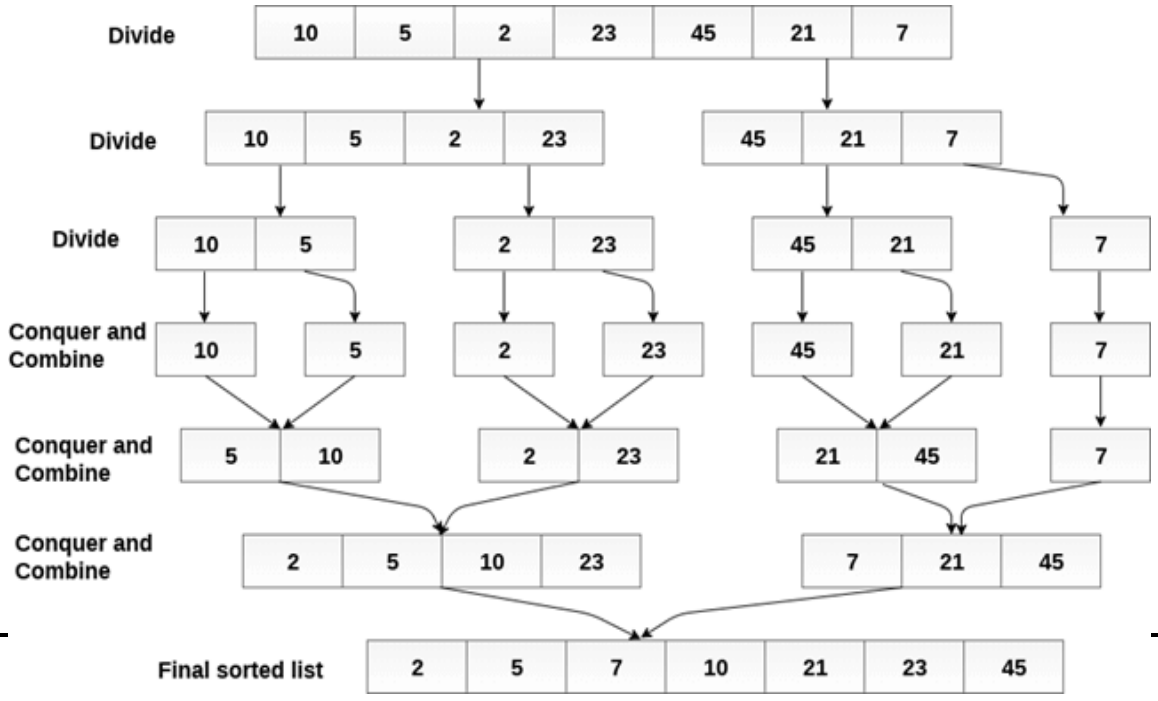
The main idea behind merge sort is that, the short list takes less time to be sorted.

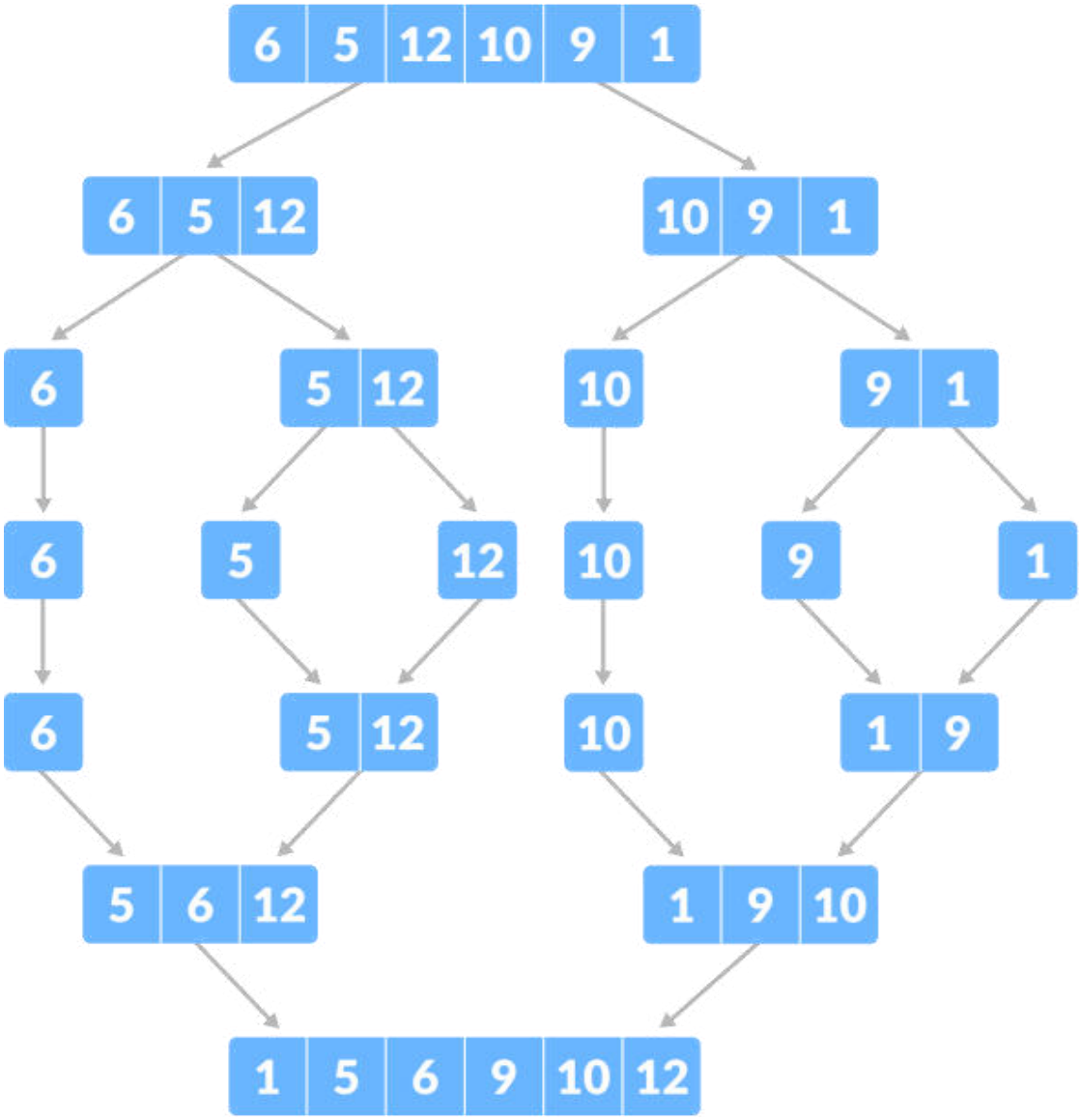
In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sortedarray is produced.

Example :

Consider the following array of 7 elements. Sort the array by using merge sort.

1. A = {10, 5, 2, 23, 45, 21, 7}





➢ QUICK SORT:

Quick Sort is an algorithm based on the DIVIDE-AND-CONQUER paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the list getscompletely sorted. The time complexity of this algorithm is O (n log n).

- Auxiliary space used in the average case for implementing recursive function calls isO (log n) and hence proves to be a bit space costly, especially when it comes to largedata sets.
- 2
- Its worst case has a time complexity of O (n) which can prove very fatal for large
  - data sets. Competitive sorting algorithms

Algorithm for Quick sort

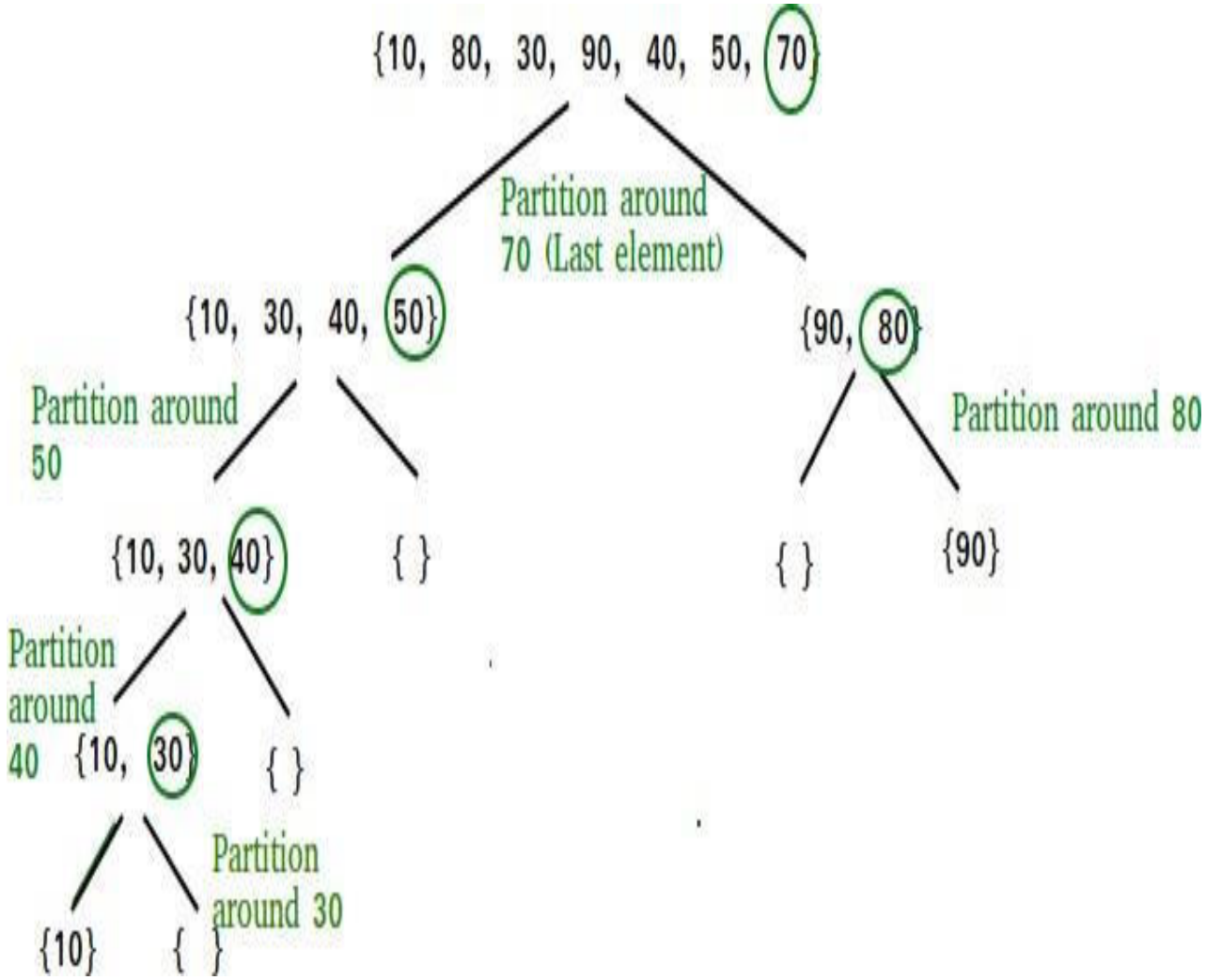
Algorithm quickSort (a, low,

```
 high)
 {
 If(high>low) then
 {
 m=partition(a,low,high);
 if(low<m) then
  quick(a,low,m);
 if(m+1<high) then
 quick(a,m+1,high);
 }
 }
 Algorithm partition(a, low,
 igh)
 {
 i=low,j=high;
  mid=(low+high)/2;
 pivot=a[mid];
  while(i<=j)
 do
 {
 while(a[i]<=pivot)
  i++;
 while(a[j]>pivot)
 j--;
 if(i<=j)
 {
 temp=a[i];
a[i]=a[j];
a[j]=temp;
i++;
j--;
}
}
return j;
}
```

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

1. Always pick first element as pivot.
2. ***Always pick last element as pivot (implemented below)***
3. Pick a random element as pivot.
4. ***Pick median as pivot.***

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



➢ Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in orderto search an element into some list by using binary search technique, we must ensure that the listis sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halvesand the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves dependingupon the result produced through the match.

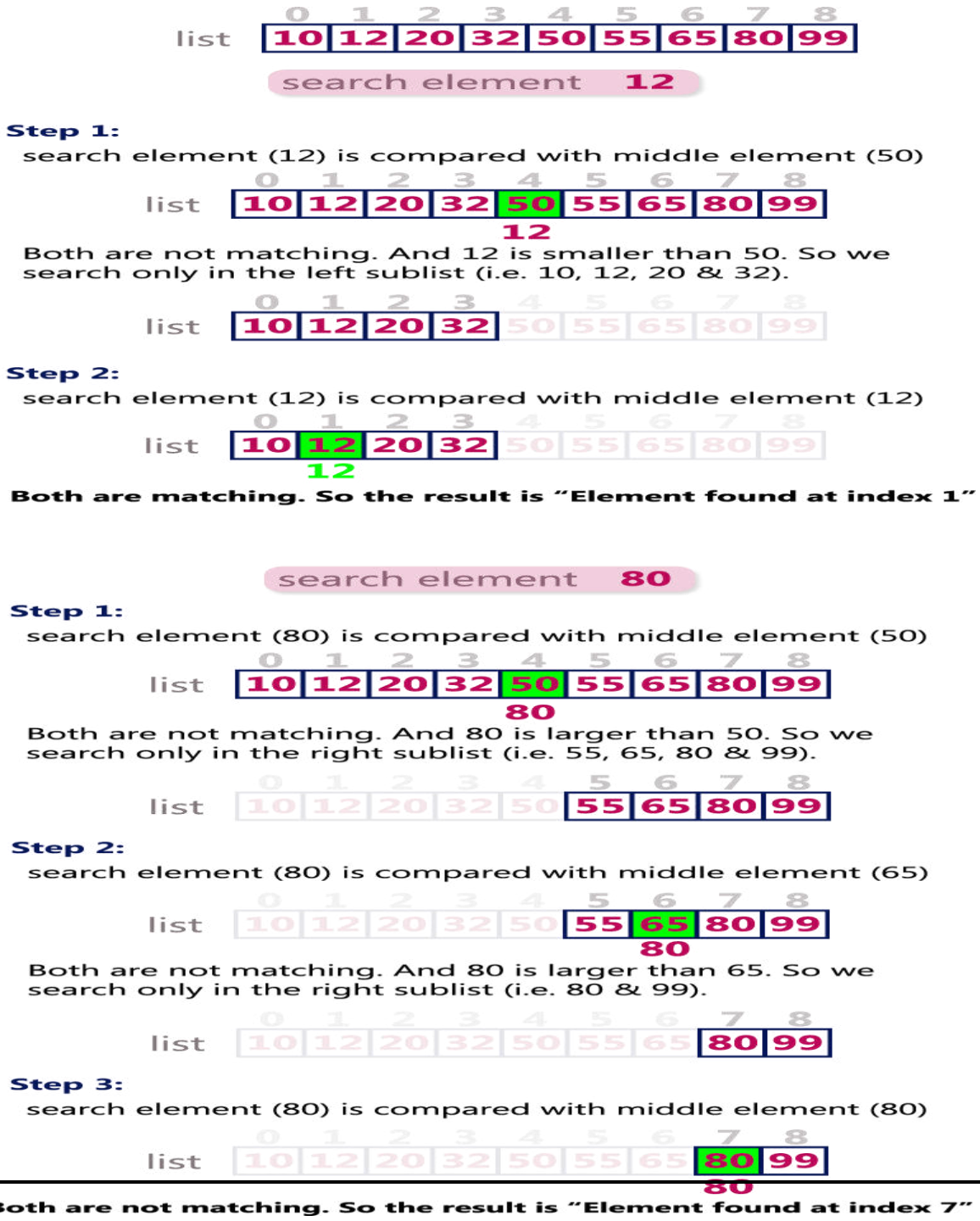
Binary search is implemented using following steps...

- **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 thefunction.
- **Step 5 -** If both are not matched, then check whether the search element is smaller orlarger than the middle element.
- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5for 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 forthe right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or untilsublist 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...



DYNAMIC PROGRAMMING:

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The sub-problems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler sub-problems, solving each sub-problem just once, and then storing their solutions to avoid repetitive computations.

Dynamic Programming is the most powerful design technique for solving optimization problems.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblems)
4. Construct the optimal solution for the entire problem form the computed values of smaller subproblems.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

CHARACTERISTICS OF DYNAMIC PROGRAMMING

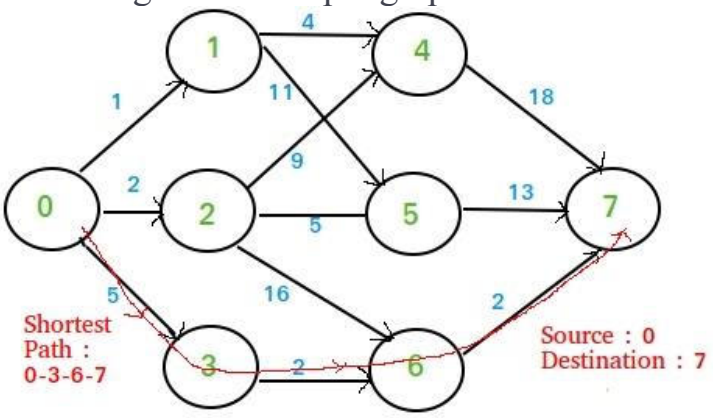
Dynamic Programming

- Dynamic programming is a technique for solving problems with a **recursive structure** with the following characteristics:
  1. **optimal substructure** (principle of optimality): An optimal solution to a problem can be decomposed into optimal solutions for subproblems, This is the defining characteristic of problems solving by DP. Not every problem has this property.
  2. **a small number of subproblems**: The total number of sub-instances to be solved is small
  3. **overlapping subproblems**: During the computation same instances are referred to over and over again

➢ MULTISTAGE GRAPH (SHORTEST PATH)

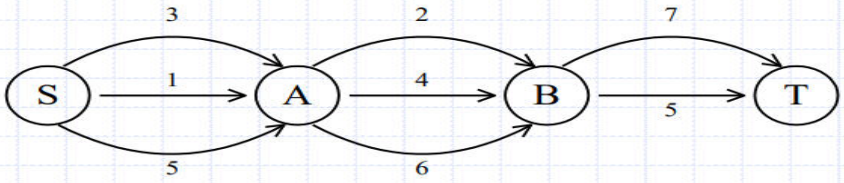
A **Multistage graph** is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage).
We are give a multistage graph, a source and a destination, we need to find shortest path from source to destination. By convention, we consider source at stage 1 and destination as last stage.

Following is an example graph we will consider in this article :-



The shortest path

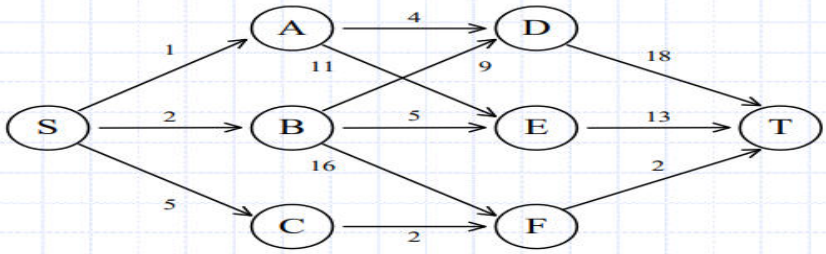
- To find a shortest path in a multi-stage graph



- Apply the greedy method :
  the shortest path from S to T :
  1 + 2 + 5 = 8

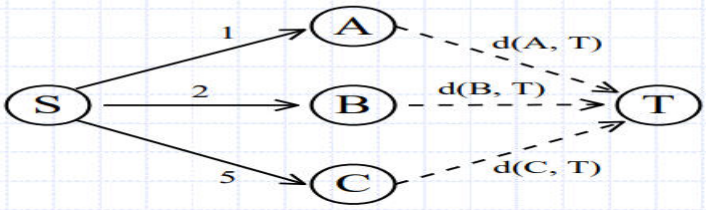
The shortest path in multistage graphs

- e.g.

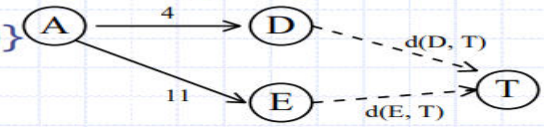


- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:
  (S, C, F, T) 5+2+2 = 9.

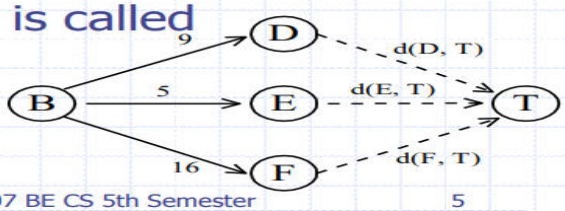
- Dynamic programming approach (forward approach):



- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
- d(A,T) = min{4+d(D,T), 11+d(E,T)}
  = min{4+18, 11+13} = 22.



- d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
  = min{9+18, 5+13, 16+2} = 18.
- d(C, T) = min{ 2+d(F, T) } = 2+2 = 4
- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
  = min{1+22, 2+18, 5+4} = 9.
- The above way of reasoning is called backward reasoning.



Backward approach (forward reasoning)

- d(S, A) = 1
  d(S, B) = 2
  d(S, C) = 5
- d(S,D)=min{d(S, A)+d(A, D),d(S, B)+d(B, D)}
  = min{ 1+4, 2+9 } = 5
  d(S,E)=min{d(S, A)+d(A, E),d(S, B)+d(B, E)}
  = min{ 1+11, 2+5 } = 7
  d(S,F)=min{d(S, A)+d(A, F),d(S, B)+d(B, F)}
  = min{ 2+16, 5+2 } = 7

- d(S,T) = min{d(S, D)+d(D, T),d(S,E)+
  d(E,T), d(S, F)+d(F, T)}
  = min{ 5+18, 7+13, 7+2 }
  = 9

Principle of optimality

- Principle of optimality: Suppose that in solving a problem, we have to make a sequence of decisions D_1, D_2, …, D_n. If this sequence is optimal, then the last k decisions, $1 < k < n$ must be optimal.
- e.g. the shortest path problem
  If i, i_1, i_2, …, j is a shortest path from i to j, then i_1, i_2, …, j must be a shortest path from i_1 to j
- In summary, if a problem can be described by a multistage graph, then it can be solved by dynamic programming.

- Forward approach and backward approach:
  - Note that if the recurrence relations are formulated using the forward approach then the relations are solved backwards . i.e., beginning with the last decision
  - On the other hand if the relations are formulated using the backward approach, they are solved forwards.
- To solve a problem by using dynamic programming:
  - Find out the recurrence relations.
  - Represent the problem by a multistage graph.

➢ OPTIMAL BINARY SEARCH TREES

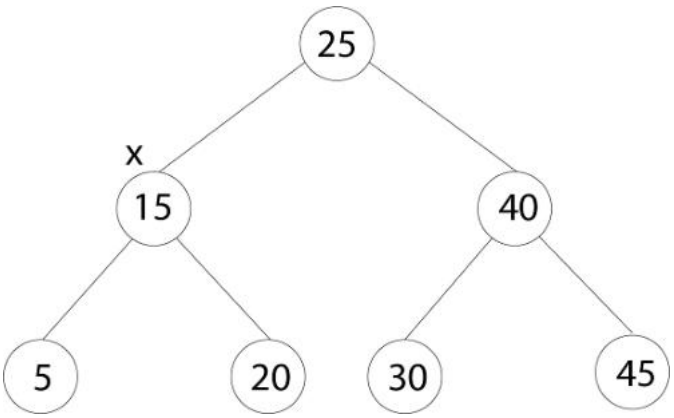
Binary Search Trees

A Binary Search tree is organized in a Binary Tree. Such a tree can be defined by a linked data structure in which a particular node is an object. In addition to a key field, each node contains field left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is Nil.

Binary Search Tree Property

Let x be a node in a binary search tree.

- If y is a node in the left subtree of x, then key [y] ≤key [k].
- If z is a node in the right subtree of x, then key [x] ≤ key [y].



NOTE: SEET THE EXAMPLE IN YOUR NOTES FOR OPTIMAL BINARY SEARCH TREE.

➢ 0/1 KNAPSACK PROBLEM

0/1 Knapsack Problem: Dynamic Programming Approach:

Knapsack Problem:

Knapsack is basically means bag. A bag of given capacity.

We want to pack n items in your luggage.

- The ith item is worth v_i dollars and weight w_i pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.
- v_i w_i W are integers.

1. W ≤ capacity
2. Value ← Max

Input:

Knapsack of capacity

List (Array) of weight and their corresponding value.

Output: To maximize profit and minimize weight in capacity.

The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

Knapsack problem can be further divided into two parts:

1. Fractional Knapsack: Fractional knapsack problem can be solved by **Greedy Strategy** where as 0 /1 problem is not.

It cannot be solved by **Dynamic Programming Approach.**

0/1 Knapsack Problem:

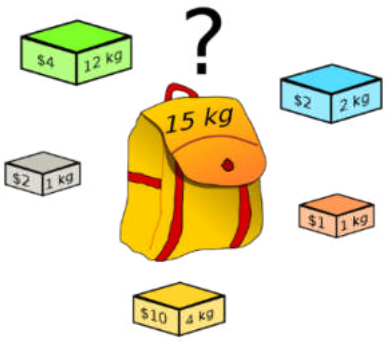
In this item cannot be broken which means thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack Problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity.
- **Greedy Approach** doesn't ensure an Optimal Solution.

The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem

0/1 Knapsack Problem Using Dynamic Programming-

Consider-

Knapsack weight capacity = w

Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01:

Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.

Fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown-

	0	1	2	3	.....	W
0	0	0	0	0	.....	0
1	0					
2	0					
	.....					
n	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right.

Use the following formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

This step leads to completely filling the table.

- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step-03:

To identify the items that must be put into the knapsack to obtain that maximum profit,

- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Time Complexity-

- Each entry of the table requires constant time $\theta(1)$ for its computation.
- It takes $\theta(nw)$ time to fill $(n+1)(w+1)$ table entries.
- It takes $\theta(n)$ time for tracing the solution since tracing process traces the n rows.
- Thus, overall $\theta(nw)$ time is taken to solve 0/1 knapsack problem using dynamic programming.

PRACTICE PROBLEM BASED ON 0/1 KNAPSACK PROBLEM-

Problem-

For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

OR

Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-

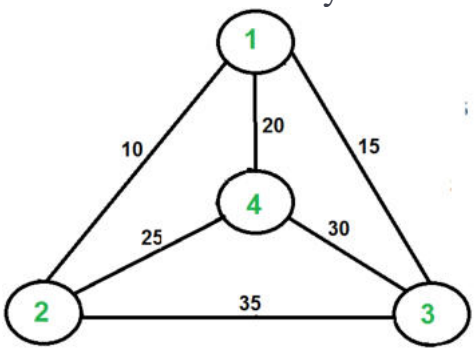
$$n = 4$$

$$w = 5 \text{ kg}$$

$$(w1, w2, w3, w4) = (2, 3, 4, 5)$$

$$(b1, b2, b3, b4) = (3, 4, 5, 6)$$

- **Travelling Salesman Problem (TSP):**
  Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
  Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.
Following are different solutions for the traveling salesman problem.

Naive Solution:
1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.
Time Complexity: $\Theta(n!)$

Dynamic Programming:
Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we

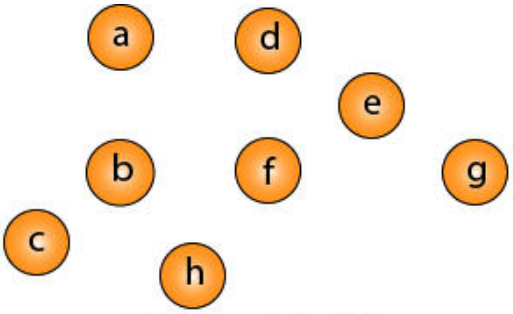
return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)?
To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*.
We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

Traveling-salesman Problem EXAMPLE

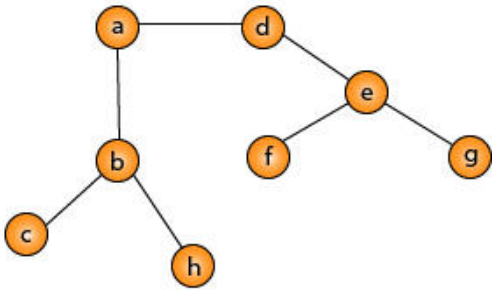
In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

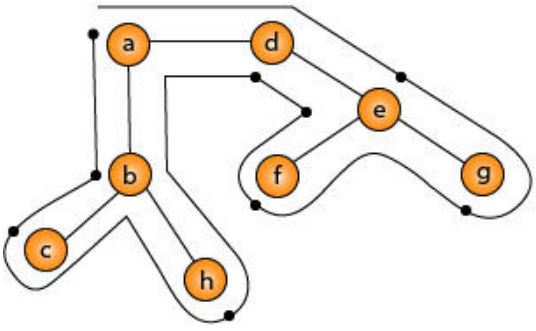
It can be shown that TSP is NPC.



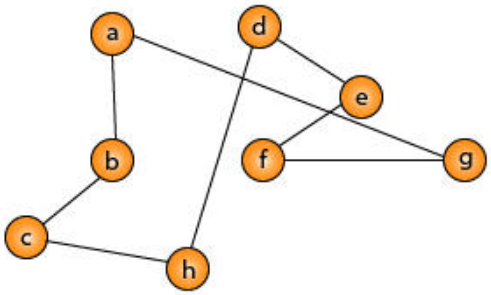
(1) A given set of points



(2) MST T



(3) Full tree walk on T.



(4) A preorder sequence gives a tour H.

Intuitively, Approx-TSP first makes a full walk of MST T, which visits each edge exactly two times. To create a Hamiltonian cycle from the full walk, it bypasses some vertices (which corresponds to making a shortcut)

UNIT- IV

➢ Backtracking (General method)

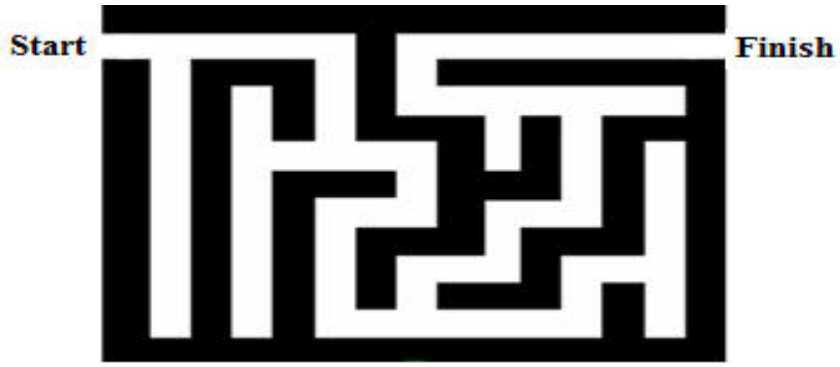
Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

Suppose you have to make a series of decisions among various choices, where

- You don't have enough information to know what to choose
- Each decision leads to a new set of choices.
- Some sequence of choices (more than one choices) may be a solution to your problem.

Backtracking is a methodical (Logical) way of trying out various sequences of decisions, until you find one that "works"

Example@1 (net example) : Maze (a tour puzzle)



Given a maze, find a path from start to finish.

- In maze, at each intersection, you have to decide between 3 or fewer choices:
  - ✓ Go straight
  - ✓ Go left
  - ✓ Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices.
- One or more sequences of choices may or may not lead to a solution.
- Many types of maze problem can be solved with backtracking.

Sorting the array of integers in a[1:n] is a problem wh ose solution is expressible by an n-tuple

x_i→ is the index in 'a' of the i^{th} smallest element.

The criterion function 'P' is the inequality $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i \leq n$

S_i→ is finite and includes the integers 1 through n.m_i→size of set S_i

$m=m_1m_2m_3---m_n$ n tuples that possible candidates for satisfying the function P.

With Brute force approach would be to form all these n-tuples, evaluate (judge) each one with P and save those which yield the optimum. By using backtrack algorithm; yield the same answer

with far fewer than 'm' trails. Many of the problems we solve using backtracking requires that all the solutions satisfy a complex set of constraints.

For any problem these constraints can be divided into two categories:

- Explicit constraints.
- Implicit constraints.

Explicit constraints: Explicit constraints are rules that restrict each x_i to take on values only from a given set.
Example: $x_i \geq 0$ or si={all non-negative real numbers}
X_i=0 or 1 or S_i={0, 1}
$l_i \leq x_i \leq u_i$ or s_i={a: $l_i \leq a \leq u_i$ }
The explicit constraint depends on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Implicit Constraints:

The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the X_i must relate to each other.

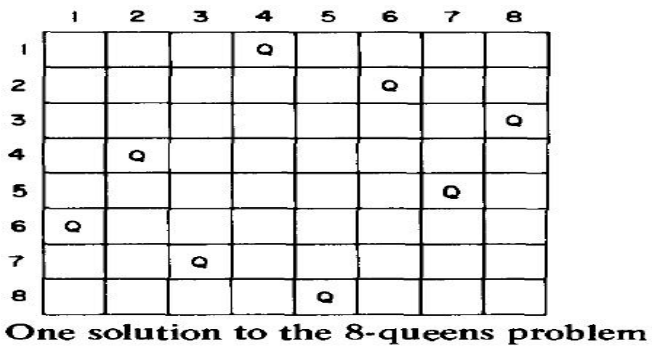
Applications of Backtracking:

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

➢ N-Queens Problem:

It is a classic combinatorial problem. The eight queen's puzzle is the problem of placing eight queens puzzle is the problem of placing eight queens on an 8×8 chessboard so that no two queens attack each other. That is so that no two of them are on the same row, column, or diagonal.

The 8-queens puzzle is an example of the more general n-queens problem of placing n queens on an n×n chessboard.



One solution to the 8-queens problem

Here queens can also be numbered 1 through 8
Each queen must be on a different row

Assume queen 'i' is to be placed on row 'i'

All solutions to the 8-queens problem can therefore be represented a s s-tuples(x_1, x_2, x_3—x_8)

xi→ the column on which queen 'i' is placed

si→{1, 2, 3, 4, 5, 6, 7, 8}, $1 \le i \le 8$

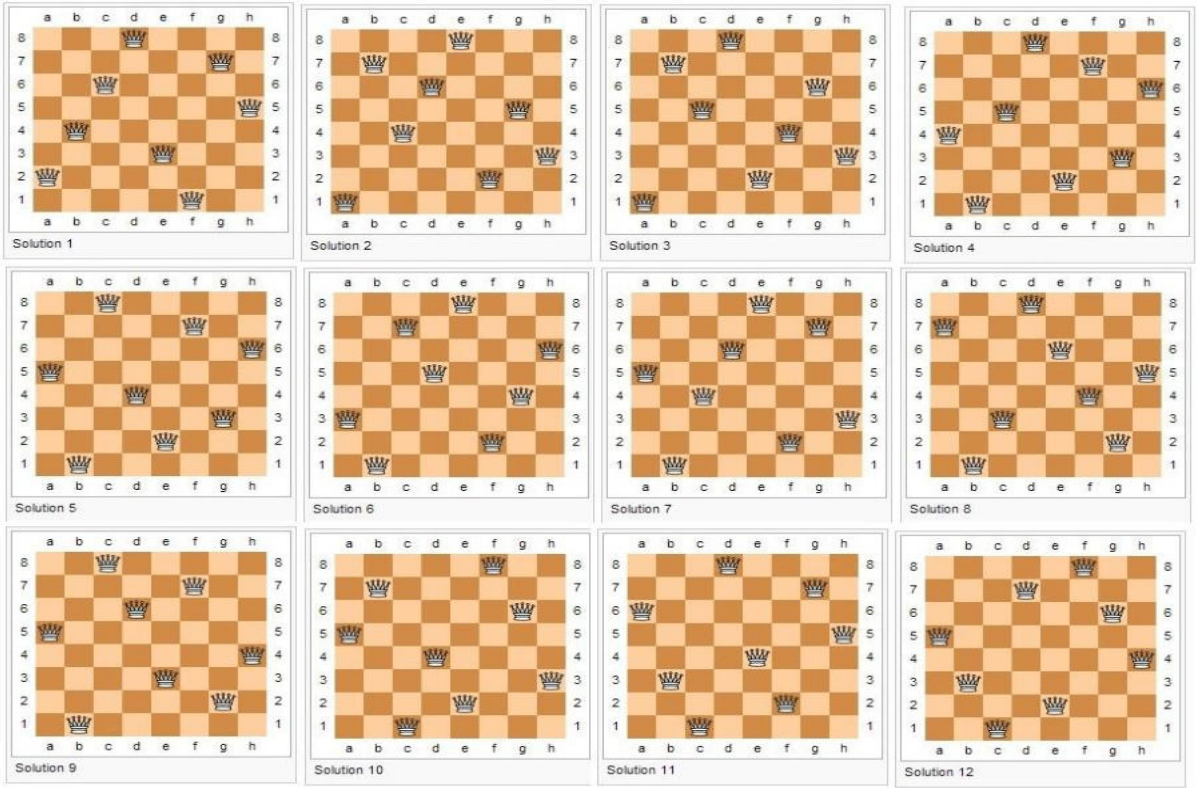
Therefore the solution space consists of 8^8 s-tuples.

The implicit constraints for this problem are that no two x_i's can be the same column and no two queens can be on the same diagonal.

By these two constraints the size of solution pace reduces from 88 tuples to 8! Tuples.

For example s_i(4,6,8,2,7,1,3,5)

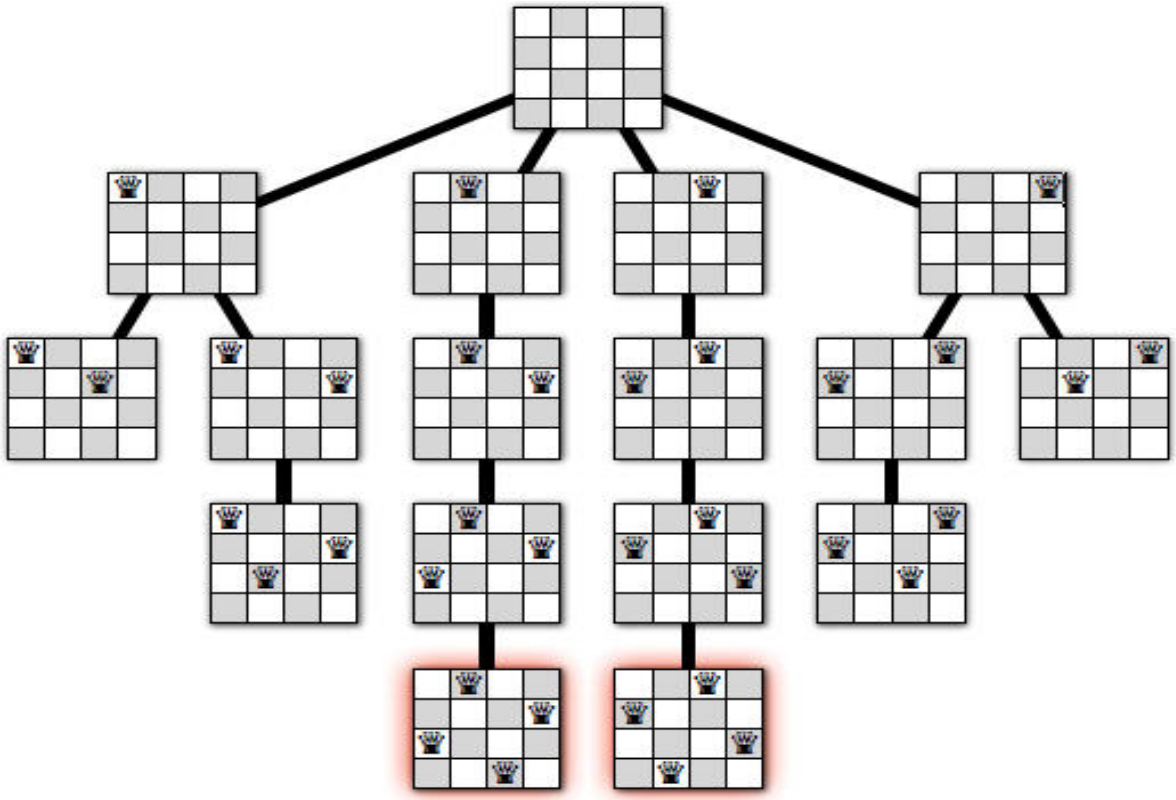
In the same way for n-queens are to be placed on an n×n chessboard, the solution space consists of all n! Permutations of n-tuples (1,2,----n).



Some solution to the 8-Queens problem

Algorithm for new queen be placed	**All solutions to the n·queens problem**

Algorithm Place(k,i)
//Return true if a queen can be placed in kth row & ith column
//Other wise return false
{
for j:=1 to k-1 do
if(x[j]=i or Abs(x[j]-i)=Abs(j-k)))
then return false
return true
}
	Algorithm NQueens(k, n)
	// its prints all possible placements of n-queens on an n×n chessboard.
	{
	for i:=1 to n do{
	if Place(k,i) then
	{
	X[k]:=I;
	if(k==n) then write (x[1:n]);
	else NQueens(k+1, n);
	}
	}}



The complete recursion tree for our algorithm for the 4 queens problem.

Sum of Subsets Problem:

Given positive numbers w_i $1 \le i \le n$, & m, here sum of subsets problem is finding all subsets of w_i whose sums are m.

Definition: Given n distinct +ve numbers (usually called weights), desire (want) to find all combinations of these numbers whose sums are m. this is called sum of subsets problem.

To formulate this problem by using either fixed sized tuples or variable sized tuples.

Backtracking solution uses the fixed size tuple strategy.

For example:
If n=4 (w_1, w_2, w_3, w_4)=(11,13,24,7) and m=31.
Then desired subsets are (11, 13, 7) & (24, 7).
The two solutions are described by the vectors (1, 2, 4) and (3, 4).

In general all solution are k-tuples (x_1, x_2, x_3---x_k) $1 \le k \le n$, different solutions may have different sized tuples.

- Explicit constraints requires $x_i \in \{j / j \text{ is an integer } 1 \le j \le n\}$
- Implicit constraints requires:
  No two be the same & that the sum of the corresponding w_i's be m
  i.e., (1, 2, 4) & (1, 4, 2) represents the same. Another constraint is $x_i < x_{i+1}$ $1 \le i \le k$

W_i→ weight of item i
M→ Capacity of bag (subset)

X_i→ the element of the solution vector is either one or zero.

X_i value depending on whether the weight wi is included or not.
If X_i=1 then wi is chosen.
If X_i=0 then wi is not chosen.

$$\underbrace{\sum_{i=1}^{k} W(i)X(i)}_{\text{Total sum till now}} + \underbrace{\sum_{i=k+1}^{n} W(i)}_{\text{Still there}} \geq M$$

The above equation specify that x_1, x_2, x_3, --- x_k cannot lead to an answer node if this condition is not satisfied.

$$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$$

The equation cannot lead to solution.

$$B_k(X(1), \ldots, X(k)) = true \text{ iff } \left(\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \geq M \text{ and } \sum_{i=1}^{k} W(i)X(i) + W(k+1) \leq M \right)$$

$$s = \sum_{j=1}^{k-1} W(j)X(j), \quad \text{and} \quad r = \sum_{j=k}^{n} W(j)$$

Recursive backtracking algorithm for sum of subsets problem

Algorithm SumOfSub(s, k, r)
{
//$s = \sum_{j=1}^{k-1} W(j)X(j)$, and $r = \sum_{j=k}^{n} W(j)$
X[k]=1
If(S+w[k]=m) then write(x[1:]); // subset found.
Else if (S+w[k] + w{k+1] ≤ M)
Then SumOfSub(S+w[k], k+1, r-w[k]);
if ((S+r - w{k] ≥ M) and (S+w[k+1] ≤M)) then
{
X[k]=0;
SumOfSub(S, k+1, r-w[k]);
}
}

➢ Graph Coloring:

Let G be a undirected graph and 'm' be a given +ve integer. The graph coloring problem is assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color yet only 'm' colors are used.

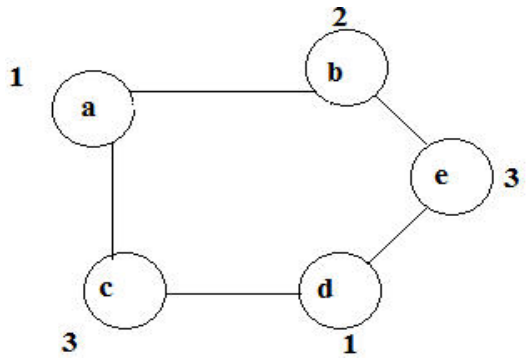
The optimization version calls for coloring a graph using the minimum number of coloring.

The decision version, known as K-coloring asks whether a graph is colourable using at most k-colors.

Note that, if 'd' is the degree of the given graph then it can be colored with 'd+1' colors.

The m- colorability optimization problem asks for the smallest integer 'm' for which the graph G can be colored. This integer is referred as "**Chromatic number**" of the graph.

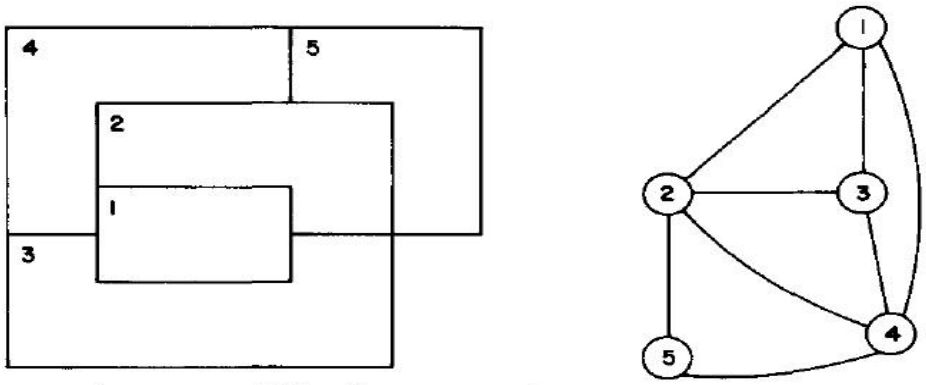
Example



- Above graph can be colored with 3 colors 1, 2, & 3.
- The color of each node is indicated next to it.
- 3-colors are needed to color this graph and hence this graph' Chromatic Number is 3.

- A graph is said to be planar iff it can be drawn in a plane (flat) in such a way that no two edges cross each other.
- **M-Colorability decision problem** is the 4-color problem for planar graphs.
- Given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only 4-colors are needed?
- To solve this problem, graphs are very useful, because a map can easily be transformed into a graph.
- Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge.

- **Example:**



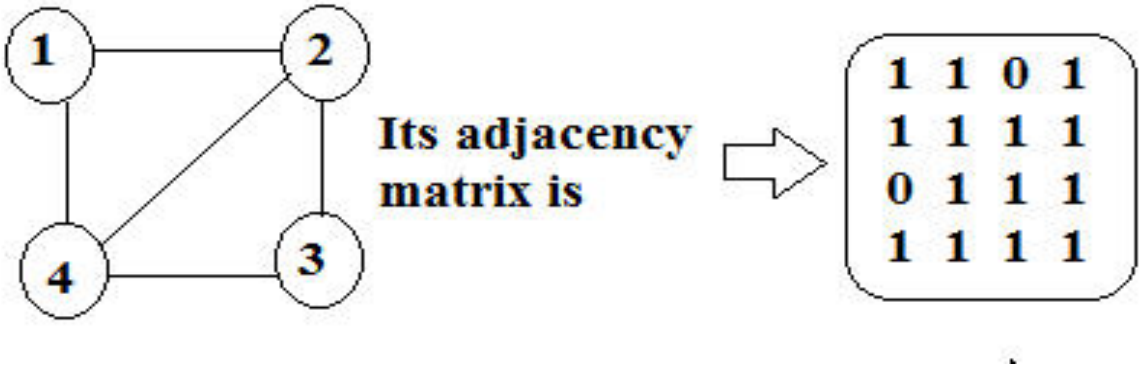
- **A map and its planar graph representation**

The above map requires 4 colors.

- Many years, it was known that 5-colors were required to color this map.
- After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They show that 4-colors are sufficient.

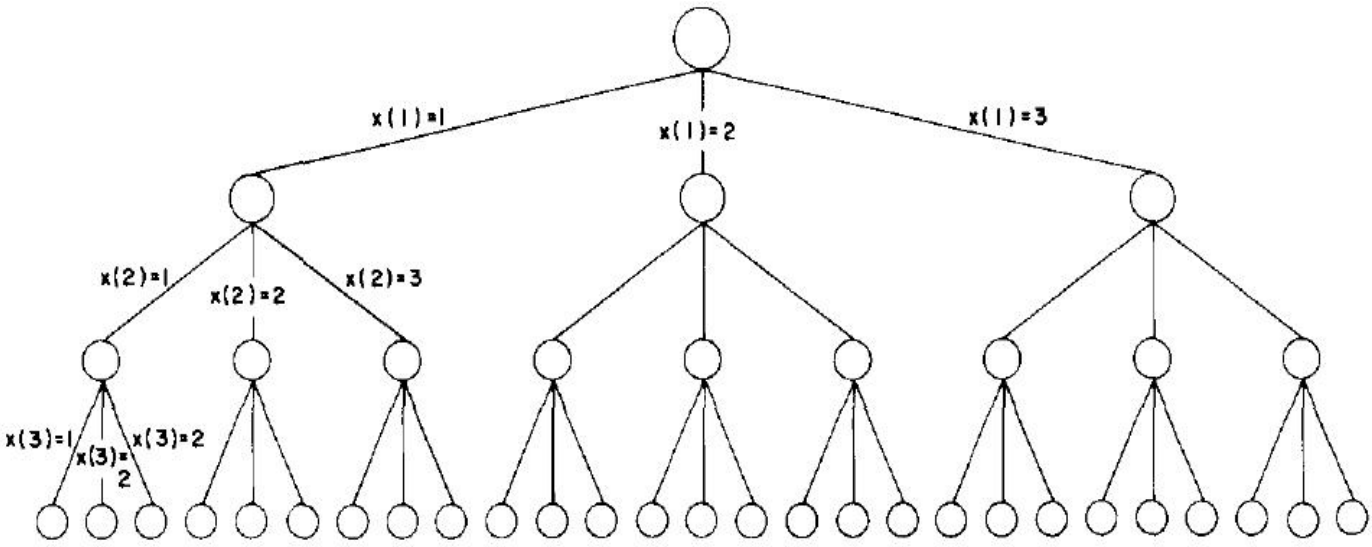
Suppose we represent a graph by its adjacency matrix G[1:n, 1:n]

Ex:



Here G[i, j]=1 if (i, j) is an edge of G, and G[i, j]=0 otherwise.
Colors are represented by the integers 1, 2,---m and the solutions are given by the n-tuple (x1, x2,---xn)
xi→ Color of node i.

State Space Tree for
n=3→ nodes
m=3→colors



State space tree for MCOLORING when $n = 3$ and $m = 3$

1st node coloured in 3-ways

2nd node coloured in 3-ways

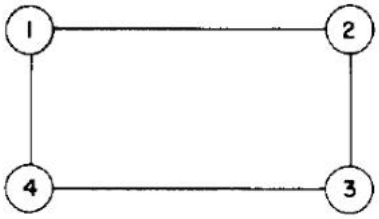
3rd node coloured in 3-ways

So we can colour in the graph in 27 possibilities of colouring.

Finding all m-coloring of a graph	Getting next color

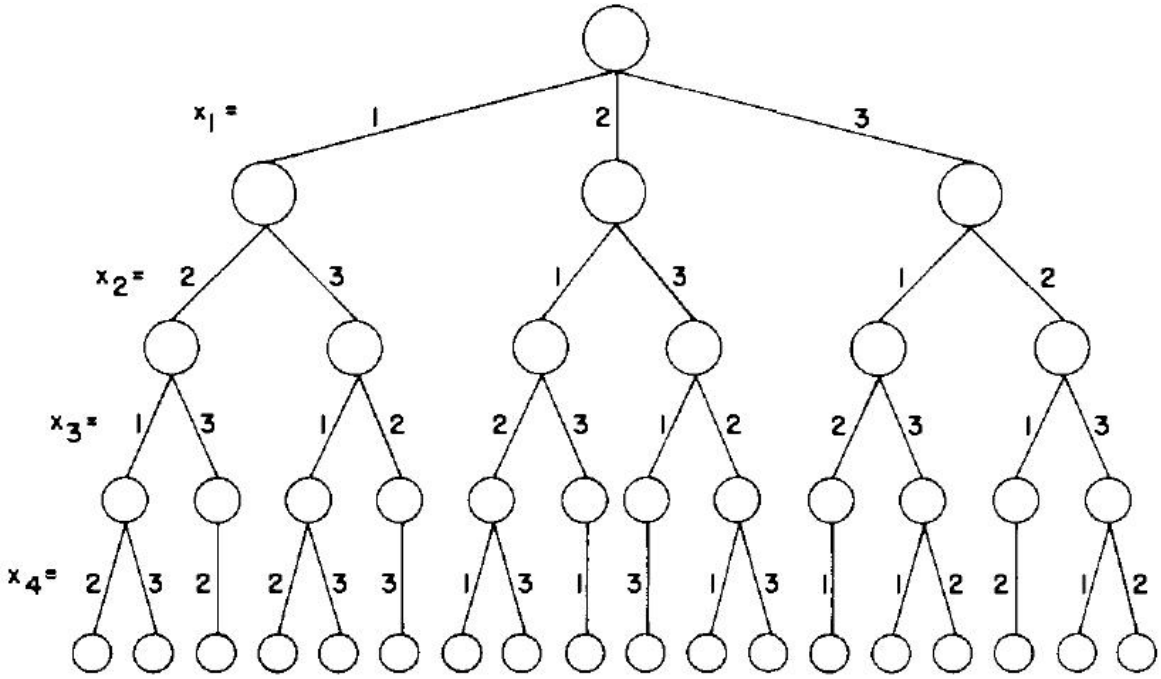
Algorithm mColoring(k){
// g(1:n, 1:n)→ boolean adjacency matrix.
// k→index (node) of the next vertex to color.
repeat{
nextvalue(k); // assign to x[k] a legal color.
if(x[k]=0) then return; // no new color possible
if(k=n) then write(x[1: n];
else mcoloring(k+1);
}
until(false)
}
	Algorithm NextValue(k){
	//x[1],x[2],---x[k-1] have been assigned integer values in the range [1, m]
	repeat {
	x[k]=(x[k]+1)mod (m+1); //next highest color
	if(x[k]=0) then return; // all colors have been used.
	for j=1 to n do
	{
	if ((g[k,j]≠0) and (x[k]=x[j]))
	then break;
	}
	if(j=n+1) then return; //new color found
	} until(false)
	}

Example:



Adjacency matrix is

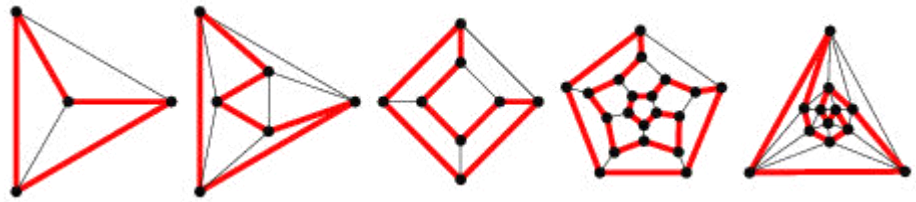
$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

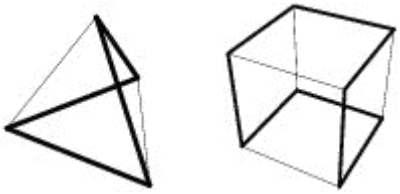


A 4 node graph and all possible 3 colorings

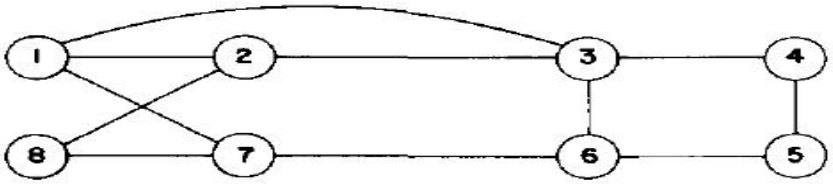
➢ Hamiltonian Cycles:

- **Def:** Let G=(V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n-edges of G that visits every vertex once & returns to its starting position.
- It is also called the Hamiltonian circuit.
- Hamiltonian circuit is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.
- A graph possessing a Hamiltonian cycle is said to be Hamiltonian graph.
    - Example:



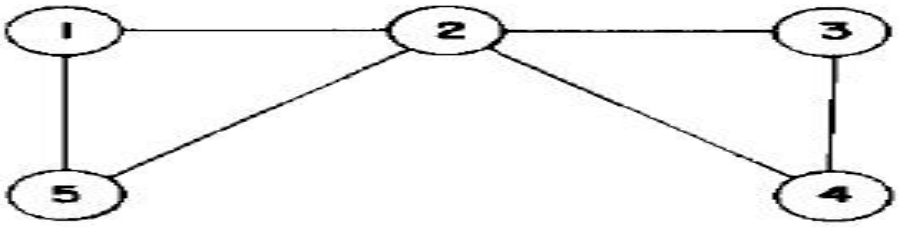


- In graph G, Hamiltonian cycle begins at some vertiex v1 $\in$ G and the vertices of G are visited in the order $v_1, v_2, \text{---} v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$.



→g1

The above graph contains Hamiltonian cycle: 1,2,8,7,6,5,4,3,1



The above graph contains no Hamiltonian cycles.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- By using backtracking method, it can be possible
- Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.
- The graph may be directed or undirected. Only distinct cycles are output.
- From graph g1 backtracking solution vector= {1, 2, 8, 7, 6, 5, 4, 3, 1}
- The backtracking solution vector (x_1, x_2, --- x_n)
    - x_i→ i^{th} visited vertex of proposed cycle.
- By using backtracking we need to determine how to compute the set of possible vertices for x_k if $x_1, x_2, x_3 \text{---} x_{k-1}$ have already been chosen.
    - If k=1 then x1 can be any of the n-vertices.

By using "NextValue" algorithm the recursive backtracking scheme to find all Hamiltoman cycles.

This algorithm is started by 1st initializing the adjacency matrix G[1:n, 1:n] then setting x[2:n] to zero & x[1] to 1, and then executing Hamiltonian (2)

Generating Next Vertex	Finding all Hamiltonian Cycles

Algorithm NextValue(k)
{
// x[1: k-1]→ is path of k-1 distinct vertices.
// if x[k]=0, then no vertex has yet been assigned to x[k]
Repeat{
X[k]=(x[k]+1) mod (n+1); //Next vertex
If(x[k]=0) then return;
If(G[x[k-1], x[k]]≠0) then
{
For j:=1 to k-1 do if(x[j]=x[k]) then break;
//Check for distinctness
If(j=k) then //if true , then vertex is distinct
If((k<n) or (k=n) and G[x[n], x[1]]≠0))
Then return ;
}
}
Until (false);
}
	Algorithm Hamiltonian(k)
	{
	Repeat{
	NextValue(k); //assign a legal next value to x[k]
	If(x[k]=0) then return;
	If(k=n) then write(x[1:n]);
	Else Hamiltonian(k+1);
	} until(false)
	}

What is a deterministic algorithm?

Deterministic algorithm is the algorithm which, given a particular input will always produce the same output, with the underlying machine always passing through the same sequence of states. In other words, Deterministic algorithm will always come up with the same result given the same inputs.

What Is A Non-deterministic Algorithm?

A nondeterministic algorithm is an algorithm that, even for same input, can exhibit different behaviors on different runs. In other words, it is an algorithm in which the result of every algorithm is not uniquely defined and result could be random.

BASIS OF COMPARISON	DETERMINISTIC ALGORITHM	NON-DETERMINISTIC ALGORITHM
Description.	Deterministic algorithm is the algorithm which, given a particular input will always produce the same output, with the underlying machine always passing through the same sequence of states.	Non-deterministic algorithm is the algorithms in which the result of every algorithm is not uniquely defined and result could be random.
Path Of Execution	In deterministic algorithm the path of execution for algorithm is same in every execution.	In Non-Deterministic algorithm the path of execution is not same for algorithm in every execution and could take any random path for its execution.
Basis Of Comparison	On the basis of execution and outcome in case of Deterministic algorithm, they are also classified as reliable algorithms as for a particular input instructions the machine will give always the same output.	Non deterministic algorithms are classified as non-reliable algorithms for a particular input the machine will give different output on different executions.
Operation	In Deterministic Algorithms execution, the target machine executes the same instruction and results same outcome which is not dependent on the way or process in which instruction get executed.	In Non-Deterministic Algorithms, the machine executing each operation is allowed to choose any one of these outcomes subjects to a determination condition to be defined later.
Output	As outcome is known and is consistent on different executions so deterministic algorithm takes polynomial time for their execution.	As outcome is not known and is non-consistent on different executions so Non-Deterministic algorithm could not get executed in polynomial time.

What Is NP Problem?

These are the decision problems which can be verified in polynomial time. That means, if I claim that there is a polynomial time solution for a particular problem, you ask me to prove it. Then, I will give you a proof which you can easily verify in polynomial time. These kinds of problems are called NP problems. Note that, here we are not talking about whether there is a polynomial time solution for this problem or not.

But we are talking about verifying the solution to a given problem in polynomial time.

To answer the rest of question, you first need to understand which NP-hard problems are also NP-complete. If an NP-hard problem belongs to set NP, then it is NP-complete. To belong to set NP, a problem needs to be

(i) A decision problem,
(ii) The number of solutions to the problem should be finite and each solution should be of polynomial length, and
(iii) Given a polynomial length solution, we should be able to say whether the answer to the problem is yes/no.

Difference Between Deterministic And Non-deterministic Algorithms

BASIS OF COMPARISON	**NP HARD PROBLEM**	**NP COMPLETE PROBLEM**
Description	NP-Hard problems (say X) can be solved if and only if there is a NP-Complete problem (say Y) can be reducible into X in polynomial time.	NP-Complete problems can be solved by deterministic algorithm in polynomial time.
Solution	To solve this problem, it must be a NP problem.	To solve this problem, it must be both NP and NP-hard problem.
Nature Of Problem	It is not a decision problem.	It is exclusively a decision problem.
Examples	-Halting problem -Vertex cover problem -Circuit-satisfiability problem etc.	-Minesweeper problem - Determining whether a graph has a Hamiltonian cycle. -Determining whether Boolean formula is satisfiable or not

Basic concepts:

NP➔ Nondeterministic Polynomial time

The problems has best algorithms for their solutions have "Computing times", that cluster into two groups

Group 1	Group 2
➢ Problems with solution time bound by a polynomial of a small degree. ➢ It also called "Tractable Algorithms" ➢ Most Searching & Sorting algorithms are polynomial time algorithms ➢ **Ex:** Ordered Search **(O (log n)),** Polynomial evaluation **O(n)** Sorting **O(n.log n)**	➢ Problems with solution times not bound by polynomial (simply non polynomial) ➢ These are hard or intractable problems ➢ None of the problems in this group has been solved by any polynomial time algorithm ➢ **Ex:** Traveling Sales Person **$O(n^2 2^n)$** Knapsack **$O(2^{n/2})$**

No one has been able to develop a polynomial time algorithm for any problem in the 2^{nd} group (i.e., group 2)

So, it is compulsory and finding algorithms whose computing times are greater than polynomial very quickly because such vast amounts of time to execute that even moderate size problems cannot be solved.

Theory of NP-Completeness:

Show that may of the problems with no polynomial time algorithms are computational time algorithms are computationally related.

There are two classes of non-polynomial time problems

1. NP-Hard
2. NP-Complete

NP Complete Problem: A problem that is NP-Complete can solved in polynomial time if and only if (iff) all other NP-Complete problems can also be solved in polynomial time.

NP-Hard: Problem can be solved in polynomial time then all NP-Complete problems can be solved in polynomial time.

All NP-Complete problems are NP-Hard but some NP-Hard problems are not know to be NP-Complete.

Nondeterministic Algorithms:

Algorithms with the property that the result of every operation is uniquely defined are termed as deterministic algorithms. Such algorithms agree with the way programs are executed on a computer.

Algorithms which contain operations whose outcomes are not uniquely defined but are limited to specified set of possibilities. Such algorithms are called nondeterministic algorithms.

The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later.

To specify nondeterministic algorithms, there are 3 new functions.

Choice(S)→ arbitrarily chooses one of the elements of sets S

Failure ()→ Signals an Unsuccessful completion

Success ()→ Signals a successful completion.

Example for Non Deterministic algorithms:

Algorithm Search(x){
//Problem is to search an element x
//output J, such that A[J]=x; or J=0 if x is not in A
J:=Choice(1,n);
if(A[J]:=x) then {
Write(J);
Success();
}
else{
write(0);
failure();
}
	Whenever there is a set of choices that leads to a successful completion then one such set of choices is always made and the algorithm terminates.
	
	A Nondeterministic algorithm terminates unsuccessfully if and only if (iff) there exists no set of choices leading to a successful signal.

Nondeterministic Knapsack algorithm	

Algorithm DKP(p, w, n, m, r, x){
W:=0;
P:=0;
for i:=1 to n do{
x[i]:=choice(0, 1);
W:=W+x[i]*w[i];
P:=P+x[i]*p[i];
}
if((W>m) or (P<r)) then Failure();
else Success();
}
	p→ given Profits
	w→ given Weights
	n→ Number of elements (number of p or w)
	m→ Weight of bag limit
	P→Final Profit
	W→Final weight

The Classes NP-Hard & NP-Complete:

For measuring the complexity of an algorithm, we use the input length as the parameter. For example, An algorithm A is of polynomial complexity p() such that the computing time of A is O(p(n)) for every input of size n.

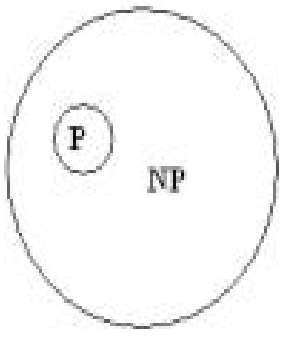
Decision problem/ Decision algorithm: Any problem for which the answer is either zero or one is decision problem. Any algorithm for a decision problem is termed a decision algorithm.

Optimization problem/ Optimization algorithm: Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

P→ is the set of all decision problems solvable by deterministic algorithms in polynomial time.

NP→ is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Since deterministic algorithms are just a special case of nondeterministic, by this we concluded that **P ⊆ NP**



Commonly believed relationship between P & NP

The most famous unsolvable problems in Computer Science is Whether P=NP or P≠NP
In considering this problem, s.cook formulated the following question.

If there any single problem in NP, such that if we showed it to be in 'P' then that would imply that P=NP.

Cook answered this question with

Theorem: Satisfiability is in P if and only if (iff) P=NP

→Notation of Reducibility

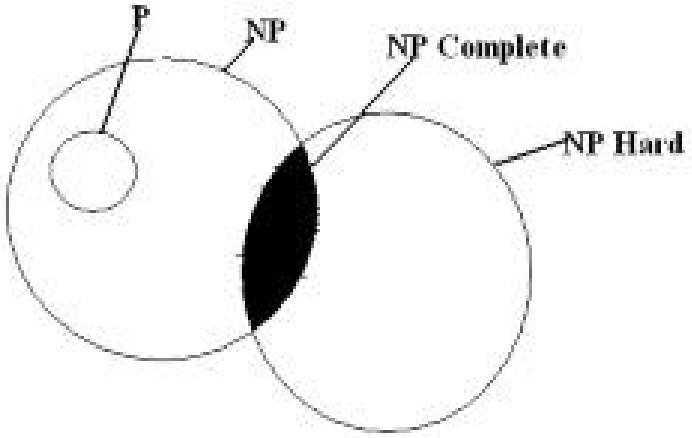
Let L_1 and L_2 be problems, Problem L_1 reduces to L_2 (written $L_1 \alpha L_2$) iff there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time

This implies that, if we have a polynomial time algorithm for L_2, Then we can solve L_1 in polynomial time.

Here α→ is a transitive relation i.e., $L_1 \alpha L_2$ and $L_2 \alpha L_3$ then $L_1 \alpha L_3$

A problem L is NP-Hard if and only if (iff) satisfiability reduces to L ie., **Statisfiability α L**

A problem L is NP-Complete if and only if (iff) L is NP-Hard and **L Є NP**



Commonly believed relationship among P, NP, NP-Complete and NP-Hard

Most natural problems in NP are either in P or NP-complete.

Examples of NP-complete problems:

- Packing problems: SET-PACKING, INDEPENDENT-SET.
- Covering problems: SET-COVER, VERTEX-COVER.
- Sequencing problems: HAMILTONIAN-CYCLE, TSP.
- Partitioning problems: 3-COLOR, CLIQUE.
- Constraint satisfaction problems: SAT, 3-SAT.
- Numerical problems: SUBSET-SUM, PARTITION, KNAPSACK.

P: is the set of decision problems that are solvable in polynomial time.

NP: is the set of decision problems that can be verified in polynomial time.

NP-Hard: L is NP-hard if for all L' ϵ NP, L' ≤p L. Thus if we can solve L in polynomial time, we can solve all NP problems in polynomial time.

NP-Complete L is NP-complete if

1. L ϵ NP and
2. L is NP-hard

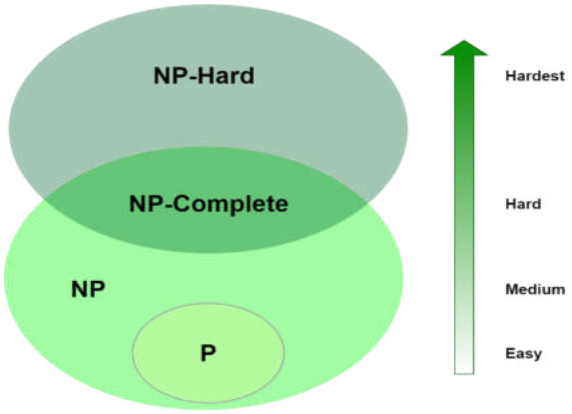
If any NP-complete problem is solvable in polynomial time, then every NP-Complete problem is also solvable in polynomial time. Conversely, if we can prove that any NP-Complete problem cannot be solved in polynomial time, every NP-Complete problem cannot be solvable in polynomial time.

Classification

let's use the same mindset that we use to classify problems in real life. While we could use a wide range of terms to classify problems, in most cases we use an "Easy-to-Hard" scale.

Now, **in theoretical computer science, the classification and complexity of common problem definitions have two major sets;** which is "Polynomial" time and which "Non-deterministic Polynomial" time.

- Easy
- Medium
- Hard
- Hardest



And we can visualize their relationship, too:

Reductions

Concept: - If the solution of NPC problem does not exist then the conversion from one NPC problem to another NPC problem within the polynomial time. For this, you need the concept of reduction. If a solution of the one NPC problem exists within the polynomial time, then the rest of the problem can also give the solution in polynomial time (but it's hard to believe). For this, you need the concept of reduction.

Example: - Suppose there are two problems, **A** and **B**. You know that it is impossible to solve problem **A** in polynomial time. You want to prove that B cannot be solved in polynomial time. So you can convert the problem **A** into problem **B** in polynomial time.

Cook's Theorem: States that satisfiability is in P if and only if P=NP
If P=NP then satisfiability is in P
If satisfiability is in P, then P=NP
To do this

- A→ Any polynomial time nondeterministic decision algorithm.
  I→Input of that algorithm
  Then formula Q(A, I), Such that Q is satisfiable iff '**A**' has a successful termination with Input **I**.
- If the length of '**I**' is '**n**' and the time complexity of **A** is **p(n)** for some polynomial **p()** then length of Q is $\mathbf{O(p^3(n) \log n)=O(p^4(n))}$
  The time needed to construct Q is also $\mathbf{O(p^3(n) \log n)}$.
- A deterministic algorithm '**Z**' to determine the outcome of '**A**' on any input '**I**'
  Algorithm **Z** computes '**Q**' and then uses a deterministic algorithm for the satisfiability problem to determine whether '**Q**' is satisfiable.
- If **O(q(m))** is the time needed to determine whether a formula of length '**m**' is satisfiable then the complexity of '**Z**' is $\mathbf{O(p^3(n) \log n + q(p^3(n) \log n))}$.
- If satisfiability is '**p**', then '**q(m)**' is a polynomial function of '**m**' and the complexity of '**Z**' becomes '**O(r(n))**' for some polynomial '**r()**'.
- Hence, if satisfiability is in **p**, then for every nondeterministic algorithm **A** in **NP**, we can obtain a deterministic **Z** in **p**.

By this we shows that satisfiability is in **p** then **P=NP**

Approximation algorithms for NP-hard problems:-

Approximation algorithms are essential tools for tackling NP-hard problems, which are problems for which no known polynomial-time algorithms can find an exact solution. Instead of finding an exact solution, approximation algorithms aim to find a solution that is close to optimal within a factor of the optimal solution, usually within polynomial time. Here's an overview of approximation algorithms, including common techniques and notable examples:

Key Concepts

1. **Approximation Ratio**:
   - The performance of an approximation algorithm is often measured by the approximation ratio, defined as the ratio between the cost of the solution found by the algorithm and the cost of the optimal solution. For a minimization problem, if C is the cost of the solution found and C* is the optimal cost, the approximation ratio ρ is given by: $\rho = C / C^*$
   -
   - For a maximization problem, the approximation ratio is: $\rho = C^*/C$
   - An algorithm with an approximation ratio of ρ is called a ρ-approximation algorithm.
2. **Polynomial Time:**
   - Approximation algorithms should run in polynomial time, making them practical for large instances of NP-hard problems.

1. **Greedy Algorithms**:
   - Greedy algorithms make a sequence of choices, each of which looks best at the moment. These algorithms are simple and often provide good approximations.
   - **Example**: The Greedy Algorithm for the Set Cover problem achieves an approximation ratio of ln n where n is the size of the universe to be covered.
2. **Local Search:**
   - Local search algorithms start with an initial solution and iteratively improve it by making local changes.
   - **Example**: The k-median problem, where local search techniques provide a constant-factor approximation.
3. **Linear Programming (LP) Relaxation:**
   - Many combinatorial optimization problems can be formulated as integer linear programs (ILPs). By relaxing the integrality constraints to allow fractional values, one can solve the resulting LP efficiently and then round the solution to an integer one.
   - **Example**: The Vertex Cover problem can be approximated using LP relaxation with a factor of 2.
4. **Primal-Dual Method:**
   - This technique involves solving both the primal and the dual linear programs simultaneously. It iteratively improves the primal and dual solutions until they converge.
   - **Example**: The primal-dual schema is used for the Facility Location problem, achieving a constant-factor approximation.
5. **Randomized Algorithms:**
   - Randomized algorithms use random choices in their logic to achieve good approximations with high probability.

- **Example**: The Max 3-SAT problem can be approximated by randomly assigning truth values, achieving an approximation ratio of 78\frac{7}{8}87.

6. **Dynamic Programming with Rounding:**
   - This technique combines dynamic programming with rounding techniques to handle fractional solutions.
   - **Example**: The Knapsack problem can be approximated using dynamic programming with rounding to achieve a polynomial-time approximation scheme (PTAS).

Notable Approximation Algorithms

1. **Traveling Salesman Problem (TSP)**:
   - For the metric TSP (where the distances satisfy the triangle inequality), the Christofides' algorithm provides a 3/2-approximation.
2. **Vertex Cover:**
   - A simple 2-approximation algorithm selects edges arbitrarily and adds both endpoints to the cover until all edges are covered.
3. **Set Cover:**
   - The greedy algorithm for Set Cover achieves an O(log n)-approximation.
4. **Max Cut:**
   - Using semidefinite programming (SDP), the Goemans-Williamson algorithm provides a 0.878-approximation.
5. **Facility Location:**
   - A combination of greedy and LP-rounding techniques results in constant-factor approximations.

Inapproximability and Hardness of Approximation

- Some NP-hard problems are hard to approximate within any reasonable factor unless P = NP. The theory of probabilistically checkable proofs (PCP) provides insights into the hardness of approximation.
- **Example**: The Set Cover problem cannot be approximated within a factor of $(1-o(1))\ln n$ unless P = NP.