**Data Structure :** A data structure is a particular way of organizing data in a computer so that it can be used effectively.

**Array:** An array is a collection of items stored at the **contiguous**(adjacent/neighboring) memory locations. It stores multiple items of the **same data type** together. Each item of array have unique index.

**String:** Strings are defined as an array of characters. The difference between a character array and string array is the string is terminated with a special character ‘\0’.

Linked List: A linked list is a linear data structure, in which the elements are n**ot stored in contiguous memory locations.** The elements in a linked list are linked using pointers. Simply liked list is a node which contains a field and link to the next node in the list.

The advantage of linked list over array is :

1. Dynamic Size
2. Ease of insertion/deletion.

**Singly Linked List**

Each node has a data and pointer to the next node.

**Head → Node 1[Data | Next] → Node 2→Node 3[Data | Next] → Null**

struct node {

int data;

struct node \*next;

}

**Doubly Linked list**

We add a pointer to the previous node in a doubly linked list. Here we can go either way, forward and backward.

**Head → prev, data, next → prev, data, next ⇾ prev, data, next →Null**

**Null ← ← <-**

struct node {

int data;

struct node \*next;

struct node \*prev;

}

**Circular Linked List**

A circular linked list is a variation of a linked list in which the last element is linked to the first element.

**Head → data, next →data, next → data, next---**

**|^---------------------------------------------------|**

**Stack**

Stack is a liner data structure which follows a particular order in which the operation are performed. The order may be **LIFO or FILO(Stack of plates).** Insertion and deletion happens on same ends.

**Queue**

A queue is a linear structure which follows a particular order in which the operations are performed. The order is **FIFO**. Insertion and deletion happens on different ends.

**Tree/Binary Tree**

A tree data structure is ano-linear and hierarchical data arranged in a tree like structure. It consists of a central node, structural nodes, sub nodes(root, branches, and leaves) which are connected by edges. Tree is a non-linear and hierarchical data structure consists of nodes that each node of tree store values. **Root node is level/depth 0.**

**Binary Search Tree**

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

* The left subtree of a node contains only nodes with **keys lesser than the node’s key**.
* The right subtree of a node contains only nodes with **keys greater than the node’s key.**
* The left and right subtree each must also be a binary search tree.
* There must be no duplicate nodes.

**Graph**

A graph data structure is a collection of nodes that have data and are connected to other nodes through edges. Like in face book everything is a **node(anything that has a data**)user, photo, event, group, page, story, video. These nodes are connected through edges.

A graph is composed of set of vertices(node/V) and set of edges(E).

Types of graphs: Null graph, Trivial graph, undirected graph, directed graph, connected graph etc.

**Searching Algorithms**

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

**Linear Searching :**

* Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
* If x matches with an element, return the index.
* If x doesn’t match with any of the elements, return -1.

**Binary Searching :**

Binary Search is a searching algorithm for finding an element's position in a **sorted array.**

* Compare x with the middle element.
* If x matches with the middle element, we return the mid-index.
* Else, If x is greater than the mid-element, then x can only lie in the right half subarray after the mid-element. So we recur for the right half.
* Else (x is smaller) recur for the left half.

**Sorting Algorithms**

A sorting algorithm is used to arrange a given array or list elements according to a comparison operator on the elements.

**Selection sort**

**Select 1st element as min number and loop through each element in the array, for i in range(len(array)) and for i in range(i+1, len(array)) #two sub arrays.**

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

A = [64, 25, 12, 22, 11]

# Traverse through all array elements

for i in range(len(A)):

m=i

for j in range(i+1,len(A)):

if A[i] > A[j]:

m=j

A[i], A[m] = A[m], A[i]

for i in range(len(A)):

print(A[i])

**Bubble sort**

**Bubble sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order**

def bubbleSort(arr):

n = len(arr)

# Traverse through all array elements

for i in range(n):

# Last i elements are already in place

#we just need to to travel till last but one element

#so n=7-0-1 = 6 (see only 6 index as index start from 0)

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

# Driver code to test above

arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

**Heaps**

**A Heap is a special Tree-based data structure in which the tree is a complete binary tree**

* Max-Heap: In a Max-Heap, the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all subtrees in that Binary Tree.
* Min-Heap: In a Min-Heap, the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all subtrees in that Binary Tree.

**Dynamic Programming**

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of sub problems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.