**Data Structure**

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

**Linear data structure**: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.

Examples of linear data structures are array, stack, queue, linked list, etc.

**Static data structure**: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.

An example of this data structure is an array.

**Dynamic data structure**: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.

Examples of this data structure are queue, stack, etc.

**Non-linear data structure**: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can’t traverse all the elements in a single run only.

Examples of non-linear data structures are trees and graphs.

**Array**

An array is a collection of items of the same variable type that are stored at contiguous memory locations. It’s one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with 0.

We can directly access an array element by using its index value.

Basic terminologies of array

Array Index: In an array, elements are identified by their indexes. Array index starts from 0.

Array element: Elements are items stored in an array and can be accessed by their index.

Array Length: The length of an array is determined by the number of elements it can contain.

**Advantages**

Arrays allow random access to elements. This makes accessing elements by position faster.

Arrays have better cache locality which makes a pretty big difference in performance.

Arrays represent multiple data items of the same type using a single name.

Arrays store multiple data of similar types with the same name.

Array data structures are used to implement the other data structures like linked lists, stacks, queues, trees, graphs, etc.

**Disadvantages**

As arrays have a fixed size, once the memory is allocated to them, it cannot be increased or decreased, making it impossible to store extra data if required. An array of fixed size is referred to as a static array.

Allocating less memory than required to an array leads to loss of data.

An array is homogeneous in nature so, a single array cannot store values of different data types.

Arrays store data in contiguous memory locations, which makes deletion and insertion very difficult to implement. This problem is overcome by implementing linked lists, which allow elements to be accessed sequentially.

**Stack**

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last

There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order

To implement the stack, it is required to maintain the pointer to the top of the stack, which is the last element to be inserted because we can access the elements only on the top of the stack.

**LIFO( Last In First Out ):**

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.

Basic Operations on Stack

In order to make manipulations in a stack, there are certain operations provided to us.

push() to insert an element into the stack

pop() to remove an element from the stack

top() Returns the top element of the stack.

isEmpty() returns true if stack is empty else false.

size() returns the size of stack.

**Queue**

A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.

**FIFO Principle of Queue:**

A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).

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(sometimes, head of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue.

**Array Representation of Queue:**

Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

Queue: the name of the array storing queue elements.

Front: the index where the first element is stored in the array representing the queue.

Rear: the index where the last element is stored in an array representing the queue.

**Linked list**

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers

In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.

Node Structure: A node in a linked list typically consists of two components:

Data: It holds the actual value or data associated with the node.

Next Pointer: It stores the memory address (reference) of the next node in the sequence.

Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

**Types of linked lists:**

There are mainly three types of linked lists:

Single-linked list

Double linked list

Circular linked list

1. **Single-linked list:**

In a singly linked list, each node contains a reference to the next node in the sequence. Traversing a singly linked list is done in a forward direction

2. **Double-linked list:**

In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.

3. **Circular linked list:**

In a circular linked list, the last node points back to the head node, creating a circular structure. It can be either singly or doubly linked

**Operations on Linked Lists**

**Insertion**: Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list

**Deletion**: Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.

**Searching**: Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

**Binary tree**

A binary tree is a tree data structure in which each node can have at most two children, which are referred to as the left child and the right child.

The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves. A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom.

Binary trees have many applications in computer science, including data storage and retrieval, expression evaluation, network routing, and game AI. They can also be used to implement various algorithms such as searching, sorting, and graph algorithms.

**Binary Tree Traversals:**

Tree Traversal algorithms can be classified broadly into two categories:

**Depth-First Search (DFS) Algorithms**

**Breadth-First Search (BFS) Algorithms**

Tree Traversal using Depth-First Search (DFS) algorithm can be further classified into three categories:

**Preorder Traversal (current-left-right):** Visit the current node before visiting any nodes inside the left or right subtrees. Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.

**Inorder Traversal (left-current-right):** Visit the current node after visiting all nodes inside the left subtree but before visiting any node within the right subtree. Here, the traversal is left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.

**Postorder Traversal (left-right-current):** Visit the current node after visiting all the nodes of the left and right subtrees. Here, the traversal is left child – right child – root. It means that the left child has traversed first then the right child and finally its root node.

Tree Traversal using Breadth-First Search (BFS) algorithm can be further classified into one category:

**Level Order Traversal:**  Visit nodes level-by-level and left-to-right fashion at the same level. Here, the traversal is level-wise. It means that the most left child has traversed first and then the other children of the same level from left to right have traversed.

**Binary Search Tree**

Itis 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.

Binary Search Tree is a node-based binary tree data structure that 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.

This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performing, a binary search is very easy.

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

There must be no duplicate nodes(BST may have duplicate values with different handling approaches)

Binary search tree

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node’s value and the right child has a value greater than the node’s value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

The root of a BST is the node that has the smallest value in the left subtree and the largest value in the right subtree. Each left subtree is a BST with nodes that have smaller values than the root and each right subtree is a BST with nodes that have larger values than the root.

**Advantages**:

Fast search: Searching for a specific value in a BST has an average time complexity of O(log n), where n is the number of nodes in the tree. This is much faster than searching for an element in an array or linked list, which have a time complexity of O(n) in the worst case.

In-order traversal: BSTs can be traversed in-order, which visits the left subtree, the root, and the right subtree. This can be used to sort a dataset.

Space efficient: BSTs are space efficient as they do not store any redundant information, unlike arrays and linked lists.

**Heap**

A Heap is a special Tree-based Data Structure in which the tree is a complete binary tree.

Types of heaps:

Generally, heaps are of two types.

**Max-Heap:**

In this heap, the value of the root node must be the greatest among all its child nodes and the same thing must be done for its left and right sub-tree also.

**Min-Heap:**

In this heap, the value of the root node must be the smallest among all its child nodes and the same thing must be done for its left and right sub-tree also.

**Properties of Heap:**

Heap has the following Properties:

Complete Binary Tree: A heap tree is a complete binary tree, meaning all levels of the tree are fully filled except possibly the last level, which is filled from left to right. This property ensures that the tree is efficiently represented using an array.

Heap Property: This property ensures that the minimum (or maximum) element is always at the root of the tree according to the heap type.

Parent-Child Relationship: The relationship between a parent node at index ‘i’ and its children is given by the formulas: left child at index 2i+1 and right child at index 2i+2 for 0-based indexing of node numbers.

**Heapify:**

It is the process to rearrange the elements to maintain the property of heap data structure. It is done when a certain node creates an imbalance in the heap due to some operations on that node. It takes O(log N) to balance the tree.

For max-heap, it balances in such a way that the maximum element is the root of that binary tree and

For min-heap, it balances in such a way that the minimum element is the root of that binary tree.

**Insertion:**

If we insert a new element into the heap since we are adding a new element into the heap so it will distort the properties of the heap so we need to perform the heapify operation so that it maintains the property of the heap.

Assume initially heap(taking max-heap) is as follows

8

/ \

4 5

/ \

1. 2

Now if we insert 10 into the heap

8

/ \

1. 5
2. / \ /
3. 2 10

After heapify operation final heap will be look like this

10

/ \

1. 8
2. / \ /
3. 2 5

**Deletion:**

If we delete the element from the heap it always deletes the root element of the tree and replaces it with the last element of the tree.

Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify operations so that it maintains the property of the heap.

Example:

Assume initially heap(taking max-heap) is as follows

15

/ \

1. 7
2. / \
3. 3

Now if we delete 15 into the heap it will be replaced by leaf node of the tree for temporary.

3

/ \

1. 7
2. /

2

After heapify operation final heap will be look like this

7

/ \

1. 3
2. /

2