DATA STRUCTURES

The way to organize and store data.

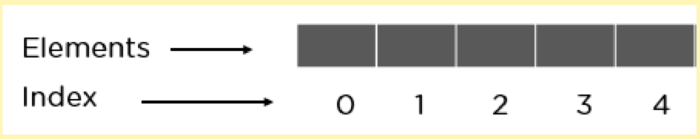
By Pratyush

Overview of Data Structures

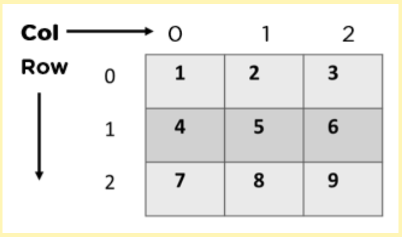
- Data Structures are the way to organize and store data in the computer system. So that It can be used efficiently.
- DS provides a meaning to manage large amount of data effectively for various applications.

Arrays

- Arrays are one of the most commonly used data structure in programming.
- Arrays are collections of homogeneous elements(i.e same data type) which stores in a contiguous memory locations, making them easy to access and manipulate.
- One Dimensional Array:



- Two Dimensional Array:



- Time Complexity:

	Access	**Search**	**Insertion**	**Deletion**
Best case →	O(1)	O(1)	O(1)	O(1)
Worst case →	O(1)	O(N)	O(N)	O(N)
Average case→	O(1)	O(N)	O(N)	O(N)

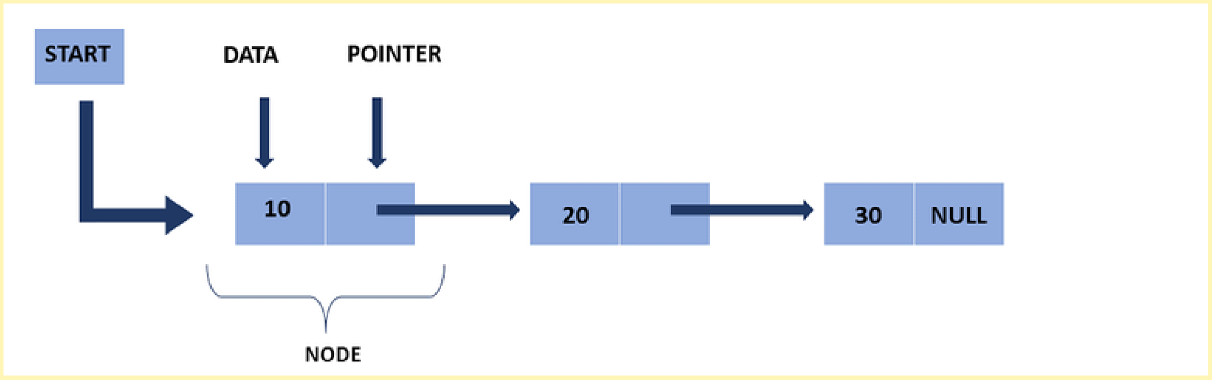
Linked Lists

- A linked list is a linear data structure that stores a collection of data elements dynamically.
- Nodes represent those data elements, and links or pointers connect each node.
- Each node consists of two fields, the information stored in a linked list and a pointer that stores the address or reference of its next node.
- The last node contains null in its second field because it will point to no node.
- A linked list can grow and shrink its size, as per the requirement i.e allowing for dynamic memory allocation and efficient insertion and deletion of elements.
- It does not waste memory space.

Types of Linked List:

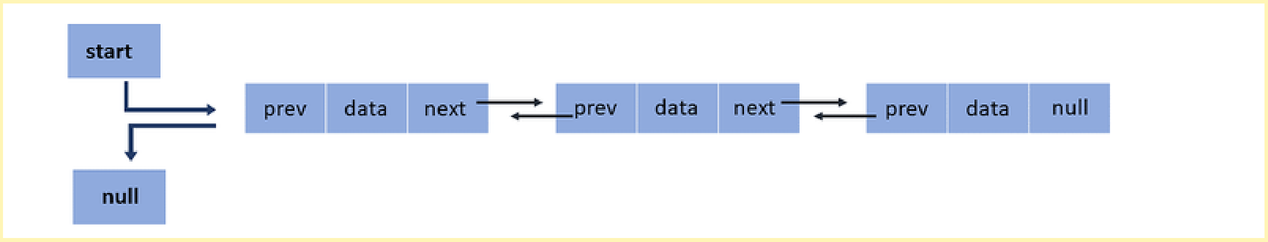
- Singly Linked List →
    - A singly linked list is the most common type of linked list. Each node has data and an address field that contains a reference to the next node.
- Time Complexity:

	Access	**Search**	**Insertion**	**Deletion**
Best case →	O(1)	O(1)	O(1)	O(1)
Worst case →	O(N)	O(N)	O(N)	O(N)
Average case→	O(N)	O(N)	O(1)	O(1)

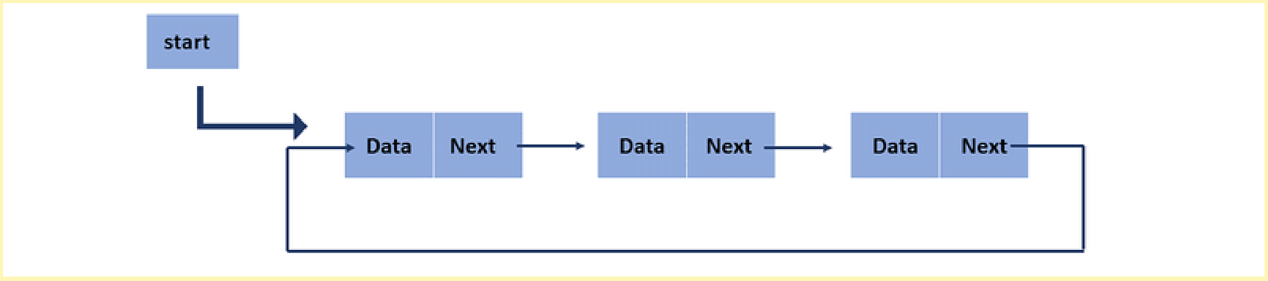


- Doubly Linked List→
    - There are two pointer storage blocks in the doubly linked list. The first pointer block in each node stores the address of the previous node. Hence, in the doubly linked inventory, there are three fields that are the previous pointers, that contain a reference to the previous node. Then there is the data, and last you have the next pointer, which points to the next node. Thus, you can go in both directions (backward and forward).
- Time Complexity:

	Access	**Search**	**Insertion**	**Deletion**
Best case →	O(1)	O(1)	O(1)	O(1)
Worst case →	O(N)	O(N)	O(1)	O(1)
Average case→	O(N)	O(N)	O(1)	O(1)

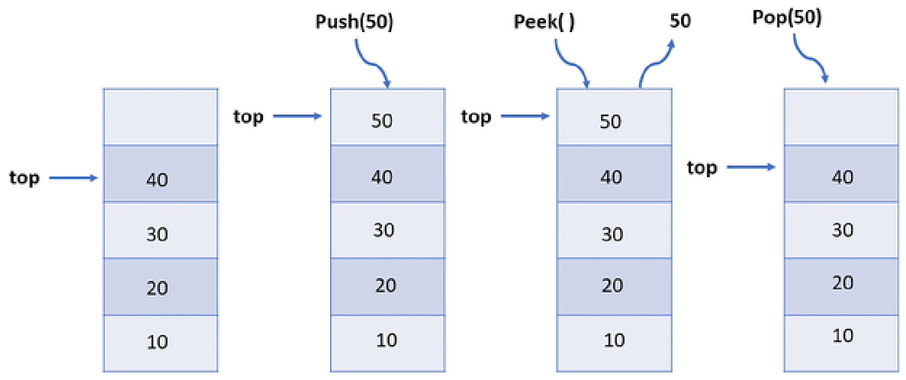


- Circular Linked List→
  - The circular linked list is extremely similar to the singly linked list. The only difference is that the last node is connected with the first node, forming a circular loop in the circular linked list.



Stacks

- The stack data structure is a linear data structure that follows a principle known as LIFO (Last In First Out)
- Real-life examples of a stack are a deck of cards, piles of books,stack of plates.
- Stacks allow for efficient insertion and deletion of elements from the top of the stacks(i.e called as Top)
- It is useful to parsing expressions and implementing recursive algorithm.
- Operations can perform over stacks are:
  - push(inserting a element into stack)
  - pop(Deletion of a element from stack)
  - peek(Retrieving the top most element from the stack)

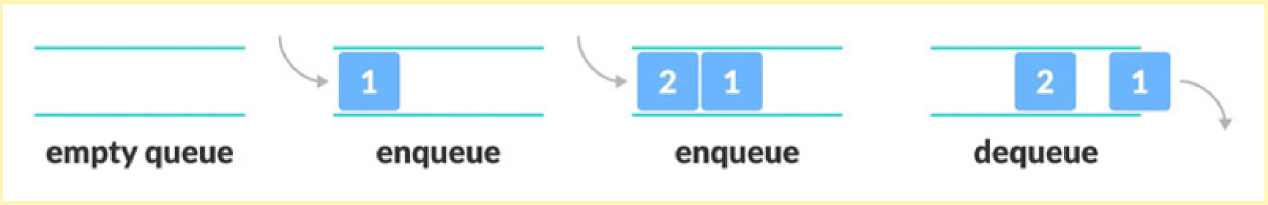


- Time Complexity:

	Access	Search	Insertion	Deletion
Best case →	O(1)	O(1)	O(1)	O(1)
Worst case →	O(N)	O(N)	O(1)	O(1)
Average case→	O(N)	O(N)	O(1)	O(1)

Queues

- Queue is a useful data structure which follows the principle of FIFO(First In First Out).
- It is usefull for task like implementation of BFS(breath First Search) algorithm, CPU scheduling, Disk Scheduling, and so on.
- Operations can perform over Queues are:
  - Enqueue: Add an element to the end of the queue
  - Dequeue: Remove an element from the front of the queue
  - IsEmpty: Check if the queue is empty
  - IsFull: Check if the queue is full
  - Peek: Get the value of the front of the queue without removing it

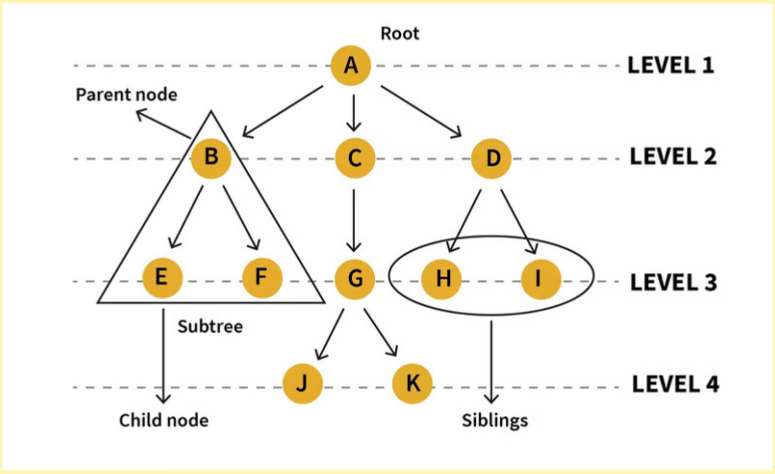


- Time Complexity:

	Access	**Search**	**Insertion**	**Deletion**
Best case →	O(1)	O(1)	O(1)	O(1)
Worst case →	O(N)	O(N)	O(1)	O(1)
Average case→	O(N)	O(N)	O(1)	O(1)

Tree

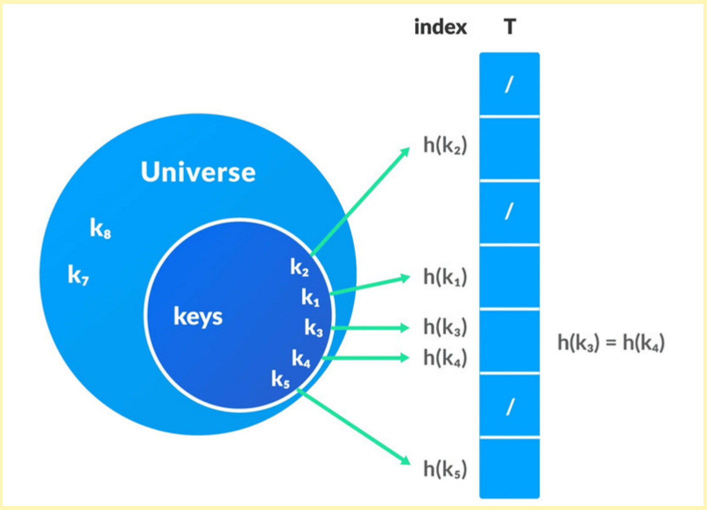
- A tree is a nonlinear hierarchical data structure which is consists of nodes connected by edges with a root node at the top.
- Why Tree?
  - Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.
  - Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.
  - Tree is useful for task such as hierarchical relationships, searching and sorting of data and implementing decision trees.



- Types of Tree
  - Binary Tree
  - Binary Search Tree
  - AVL Tree
  - B-Tree

Hash Tables

- A Hash table is a data structure that uses a hash function to map keys to values , allowing efficient lookup , insertion and deletion.
- The Hash table data structure stores elements in key-value pairs where
  - **Key**- unique integer that is used for indexing the values
  - **Value** - data that are associated with keys.
- Hash table is useful when :
  - constant time lookup and insertion is required
  - cryptographic applications
  - indexing data is required
- **Hashing (Hash Function)**
  - In a hash table, a new index is processed using the keys. And, the element corresponding to that key is stored in the index. This process is called hashing.
  - Let k be a key and h(x) be a hash function.
  - Here, h(k) will give us a new index to store the element linked with k.



- **Hash Collision**
  - When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a hash collision.
- Time Complexity:

	Access	**Search**	**Insertion**	**Deletion**
Best case →	O(1)	O(1)	O(1)	O(1)
Worst case →	O(N)	O(N)	O(N)	O(N)
Average case→	O(1)	O(1)	O(1)	O(1)