Data Structure:

- Structure of Data
  - The organization of data in memory.
  - Operations performed on that data.
- Types of data structures:
  - Linear - Array, Stack, Queue, Linked List.
  - Non-Linear - Tree, Graph
  - Associative:- data is stored in key value pair e.g. HashTable
- Usually data structures are represented as "Abstract Data Types".
- Example Data Structures:
  - Array, Stack, Queue, Linked List, Tree, Graph, Hashtable, etc.

Stack (ADT):

- Operations:
  - Push()
  - Pop()
  - Peek()
  - IsEmpty()
  - IsFull()
- Data Organization In Memory:
  - Using Arrays:
    - int info[MAX];
    - int top;
  - Using Linked List;
    - node *head;
    - Push() ==>AddFirst();
    - Pop() ==>DelFirst();
    - Peek() ==> return head->data;
    - IsEmpty() ==> return head==NULL;

Queue (ADT)

- Operations:
  - Push() //insert //addqueue
  - Pop() // deletequeue // remove
  - Peek()
  - IsEmpty()
  - IsFull()

- Data orgranization in memory:
    - Using linked list (with head & tail pointer):
        - front => node *head;
        - rear => node *tail;
        - Push() =>AddLast()
        - Pop() =>DelFirst()
        - Peek() => return head->data;
        - IsEmpty() => return head==NULL;
    - Using arrays:
        - int info[MAX];
        - int front;
        - int rear;

Types of Queues:

1. Linear Queue:
    a. No proper utilization of memory.
2. Circular Queue:
    a. Proper utilization of memory.
    b. Increments rear and front in circular fashion
        i.e. 0, 1, 2, 3, ..., MAX-1, 0, 1, ...
3. Priority Queue:
    a. Each element is associated with some priority.
    b. Element with highest priority is popped out first. (No FIFO behavior).
    c. Typically it is implemented as sorted linked list. While insertion element is added at appropriate position as per its priority. So insertion time complexity will be O(n). While deleting first (head) element is deleted (as it is element with highest priority).

4. Double Ended Queue (deque):
    a. Can push/pop elements from both sides i.e. front as well as rear.
    b. Usually implemented as doubly linked list with head & tail.

Stack Vs:Queue

1.

Stack is LIFO/FILO utility data structure.

Queue is FIFO/LILO utility data structure.

2

Stack- Push and Pop operations are done from the same end i.e. "top".

Queue- Push and Pop operations are done from different ends i.e. "rear" and "front" respectively.

3

Stack

There are no types of stack. However in single array you can implement

two or more stacks (called as multi-stack approach).

Queue

There are four types of queues i.e. linear, circular, priority and deque.

4

Applications of Stack

- Function activation records are created on the stack for each function call.
- To solve infix expression by converting to prefix or postfix.
- Parenthesis balancing
- To implement algos like Depth First Search.

Applications of **Queue**

- Printer maintains queue of documents to be printed.
- OS uses queues for many functionalities: Ready queue, Waiting queue, Message queue.
- To implement algos like Breadth First Search.

Time complexities of different algorithms:

1. Stack push & pop: T = k. O(1)
2. Queue push & pop: T = k. O(1)
3. Singly Linked list with head pointer:
    a. Add first -> T = k. O(1)
    b. Add last -> T α n O(n)
    c. Del first -> T = k. O(1)
    d. Search -> T α n / 2 O(n)
4. Singly Linked list with head & tail pointer:
    a. Add first -> T = k. O(1)
    b. Add last -> T = k. O(1)
    c. Del first -> T = k. O(1)
    d. Search -> T α n / 2 O(n)

Insertion Sort:

```
for(i=1; i<n; i++)
{

  temp = arr[i];

   for(j=i-1; j >= 0 &&arr[j] > temp; j--)
       arr[j+1] = arr[j];

   arr[j+1] = temp;
}
```

- Time complexity:
    - Best case: O(n)
    - Average/worst case: $O(n^2)$

Quick Sort:

- Time Complexity :
    - Avg case : O(n log n)
    - Worst case : $O(n^2)$

Merge Sort

- Time Complexity :
    - Avg/Worst case : O(n log n)

Heap Sort:

- Array implementation of almost complete binary tree is called as "Heap".
- Time Complexity :
    - Avg/Worst case : O(n log n)

- **Types of Linked Lists:**
  - **Singly Linear Linked List**
    - **A. Singly Linked List – only head pointer:**
      1. **Add First:O(1)**
      2. **Add Last: O(n)**
      3. **Insert at position: O(n)**
      4. **Delete First: O(1)**
      5. **Delete At Position: O(n)**
      6. **Delete All: O(n)**

    - B. **Singly Linked List –head and tail pointer:**
      1. **Add First: O(1)**
      2. **Add Last: O(1)**
      3. **Delete First: O(1)**

  - **Singly Circular Linked List:**
    Last node's next pointer keeps address of first (head) node.
    1. **Add First: O(n)**
    2. **Add Last: O(n)**
    3. **Insert at position: O(n)**
    4. **Delete First: O(n)**
    5. **Delete At Position: O(n)**

  - **Doubly Linear Linked List**
    Each node contains data, address of next node and address of previous node.
    1. **Add First: O(1)**
    2. **Add Last: O(n)**
    3. **Insert at position: O(n)**
    4. **Delete First: O(1)**
    5. **Delete At Position: O(n)**
    6. **Delete All: O(n)**

- **Doubly Circular Linked List**
  Each node contains data, address of next node and address of previous node. Last node's next contains address of first node, while first node's prev contains address of last node.
  1. **Add First: O(1)**
  2. **Add Last: O(1)**

	Singly Linked List with Head	**Singly Linked List with Head & Tail**
1.	Add last operation is slower i.e. O(n)	Add last operation is faster i.e. O(1)
2.	No overload.	Extra overhead i.e. tail pointer.
3.	Simplified list manipulation operations.	List manipulation need to consider tail pointer wherever appropriate.

	Singly Linked List	**Doubly Linked List**
1.	Unidirectional access.	Bi-directional access.
2.	For search and delete operation two pointers are required (Need to maintain prev pointer) during traversing.	For search and delete operation only one pointer is required (No need to maintain prev pointer) during traversing.
3.	Simplified list manipulation operations.	List manipulation need to consider prev pointer of each node in all operations.

	Doubly Linked List	**Doubly Circular Linked List**
1.	Traversing till last node is slower (add last operation) i.e. O(n)	Traversing till last node is faster (add last operations) i.e. O(1)