DAY-07:
- queue can be implemented by 2 ways:
1. static implementation of queue (by using an array)
2. dynamic implementation of queue (by using an linked list).

1. static implementation of queue (by using an array):

```
int arr[ 5 ];
int front;
int rear;
```

```
arr   : int []
front : int
rear  : int
```

Circular Queue:
rear = 4, front = 0
rear = 0, front = 1
rear = 1, front = 2

in a cir quueue => if front is at next pos of rear => queue full
front == (rear + 1)%SIZE

for rear = 0, front = 1 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 1 == (0+1)%5
=> 1 == 1%5
=> 1 == 1 => LHS == RHS => cir q is full

for rear = 1, front = 2 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 2 == (1+1)%5
=> 2 == 2%5
=> 2 == 2 => LHS == RHS => cir q is full

```
for rear = 2, front = 3 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 3 == (2+1)%5
=> 3 == 3%5
=> 3 == 3 => LHS == RHS => cir q is full

for rear = 3, front = 4 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 4 == (3+1)%5
=> 4 == 4%5
=> 4 == 4 => LHS == RHS => cir q is full



for rear = 4, front = 0 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 0 == (4+1)%5
=> 0 == 5%5
=> 0 == 0 => LHS == RHS => cir q is full


rear++;
rear = rear + 1;

rear = ( rear + 1 ) % SIZE

for rear=0 => rear=(rear+1)%SIZE = (0+1)%5 = 1%5 = 1
for rear=1 => rear=(rear+1)%SIZE = (1+1)%5 = 2%5 = 2
for rear=2 => rear=(rear+1)%SIZE = (2+1)%5 = 3%5 = 3
for rear=3 => rear=(rear+1)%SIZE = (3+1)%5 = 4%5 = 4
for rear=4 => rear=(rear+1)%SIZE = (4+1)%5 = 5%5 = 0
```

2. dynamic implementation of queue (by using an linked list : DCLL).

- if list is empty => queue is empty
- there is no queue full condition for dynamic queue

Enqueue => addLast()
Dequeue => deleteFirst()

OR
head => 44

Enqueue => addFirst()
Dequeue => deleteLast()

Lab Work:
1. implement dynamic queue
2. implement priority queue by using dcll

Priority Queue by using Linked List:
searchAndDelete()

```
class Node{
    int data;
    Node next;
    Node prev;
    int priorityValue;
}
```

Basic Data Structures: comfortable

Advanced Data Structures:
- tree (can be implemented by using an array as well as linked list).
- binary heap (array implementation of a tree)
- graph (array & linked list)
- hash table (array & linked list)
- merge sort & quick sort

+ Tree:
+ tree terminologies:
root node
parent node/father
child node/son
grand parent/grand father
grand child/grand son
ancestors => all the nodes which are in the path from root node to that node

- further restrictions can be applied on binary tree to mainly to achieve addition, deletion and searching operations effciently expected in O(log n) time => binary search tree can be formed.

	Addition	Deletion	Searching
Array	O(n)	O(n)	O(log n)
Linked List	O(1)	O(1)	O(n)
BST	O(log n)	O(log n)	O(log n)

- binary search tree => it is a binary tree in which left child is always smaller than its parent, and right child is always greater than or equal to its parent.

While adding node into the BST => We need to first find/search its appropriate position in a BST and after that we can add node at that position.

- there are basic two tree traversal methods:

1. bfs (breadth first search) traversal / levelwise traversal:

- traversal always starts from root node
- and nodes in a bst gets visited levelwise from left to right.

2. dfs (depth first search) traveral

- under dfs traversal further there are 3 ways by which tree can be traversed:

i. Preorder (V L R) :

- start traversal always from root node
- first visit cur node, then visit its left subtree and we can visit right subtree of any node only after either visitng its whole left subtree or left subtree is empty.
- in this traversal, root node always gets visited first, and this property remains recursively true for each subtree.

ii. Inorder (L V R) :

iii. Postorder (L R V):

Lab Work : implement recusrive addNode() function for BST.