EE4371 : ASSIGNMENT 4 : DATA STRUCTURES

Name : Kalakonda Rohini
Roll number : EE21B066

Question 1.

Given a Queue data structure that supports standard operations like **enqueue()** and **dequeue()**. The task is to implement a Stack data structure using only instances of Queue and Queue operations allowed on the instances.
A Stack can be implemented using two queues. Let Stack to be implemented be ‘s’ and queues used to implement are ‘q1’ and ‘q2’. Stack ‘s’ can be implemented in two ways:

Implement Stack using Queues By making push() operation costly:

Below is the idea to solve the problem:

*The idea is to keep newly entered element at the front of **‘q1’** so that pop operation dequeues from **‘q1’**. **‘q2’** is used to put every new element in front of **‘q1’**.*

Follow the below steps to implement the **push(s, x)** operation:

- Enqueue x to q2.
- One by one dequeue everything from q1 and enqueue to q2.
- Swap the queues of q1 and q2.

Follow the below steps to implement the **pop(s)** operation:

- Dequeue an item from q1 and return it.

Below is the implementation of the above approach.

```
# Program to implement a stack using
# two queue
from _collections import deque
```

```
class Stack:

    def __init__(self):

        # Two inbuilt queues
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x):

        # Push x first in empty q2
        self.q2.append(x)

        # Push all the remaining
        # elements in q1 to q2.
        while (self.q1):
            self.q2.append(self.q1.popleft())

        # swap the names of two queues
        self.q1, self.q2 = self.q2, self.q1

    def pop(self):

        # if no elements are there in q1
        if self.q1:
            self.q1.popleft()

    def top(self):
        if (self.q1):
            return self.q1[0]
        return None

    def size(self):
        return len(self.q1)


# Driver Code
if __name__ == '__main__':
```

```
s = Stack()
s.push(1)
s.push(2)
s.push(3)

print("current size: ", s.size())
print(s.top())
s.pop()
print(s.top())
s.pop()
print(s.top())

print("current size: ", s.size())
```

Time Complexity:

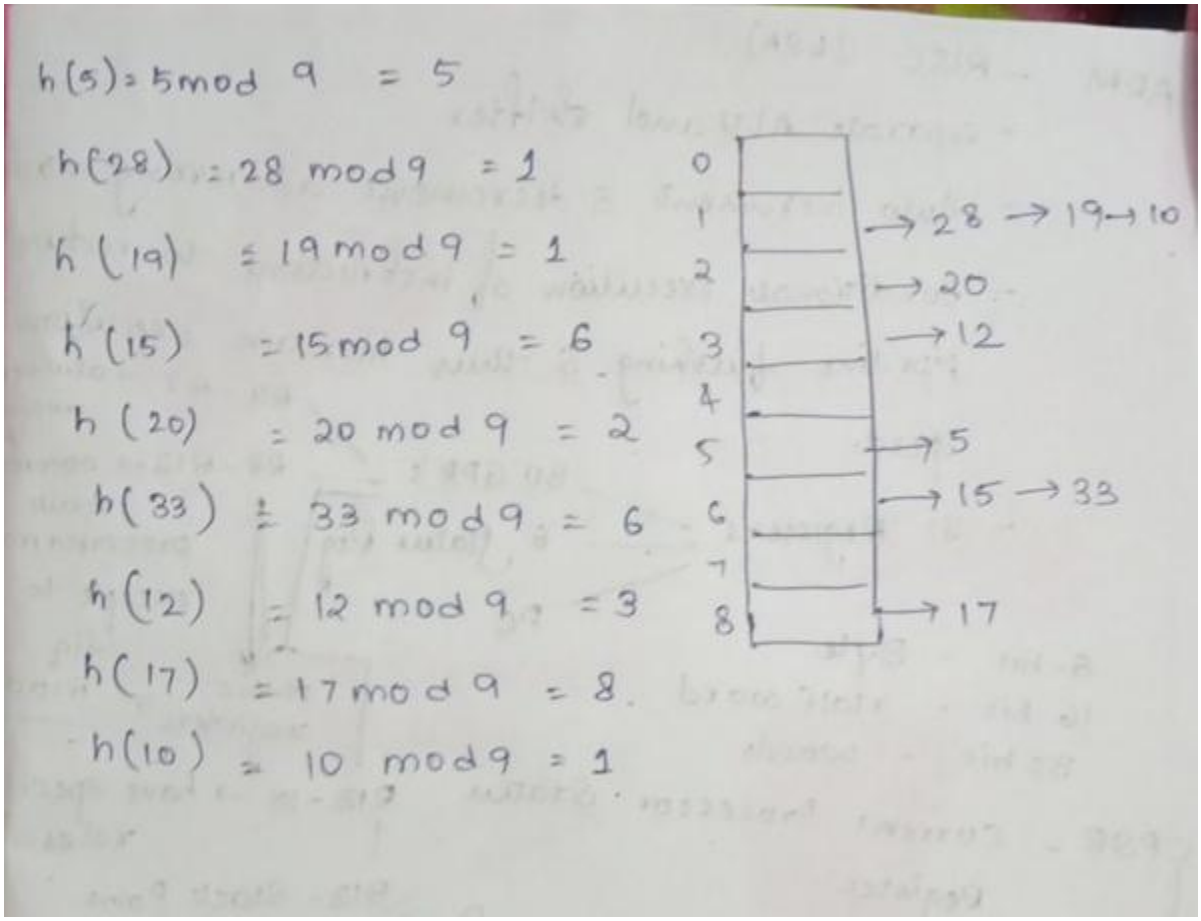
- Push operation**:** O(N), As all the elements need to be popped out from the Queue (q1) and push them back to Queue (q2).
- Pop operation: O(1), As we need to remove the front element from the Queue.

Auxiliary Space: O(N), As we use two queues for the implementation of a Stack.

Question 2 :

The given Hash table has 9 slots with the Hash Function h(k) = k mod 9. This means that the given key k will be inserted into the index given by K Mod 9. It is also mentioned that the Hash table resolves collisions by chaining. When two keys get the same index when passed through the hashing function a collision occurs. In chaining mechanism such conditions are dealt using linked list. Whenever there is a collision at a particular index of the Hash Function a linked list is generated at that index. All keys which have that particular index gets chained together as a linked list, that is the first key points to the second key, second to the third and so on. The advantage of using a linked list is that the size need not be predetermined so it can be extended to any desired length. When the keys are inserted, k mod 9 is calculated for each key and the key is inserted

into this index if a key already occupies that index, the new key is chained with the existing key to form a linked list.
Implementation :



Question 3 :
Given : A BST whose keys are distinct with a node x whose right subtree is empty.The node x is guaranteed to have a successor y.
Claim : y is the lowest ancestor of x whose left child is also an ancestor of x.
Proof :
The right subtree of x is empty and so y cannot be in the right subtree of x. If y is in the left subtree of x then y should be less than x from the property of a binary search tree. But y is greater than x because y is the successor of x. Hence y cannot be in the left subtree of x. Since y cannot be in the left subtree nor the right subtree y cannot be a descendant of x.

Now assume y is not an ancestor of x. Then let z be the lowest common ancestor of x and y. Since y is greater than x, (because y is the success of x) y has to belong to the right subtree of z and x has to belong to the left subtree of z. but

then x<z<y from the property of a binary search tree. This contradicts the fact that y is the successor of x as z lies between x and y. Hence y has to be an ancestor of x.

If y is an ancestor of x then the right side or left side of y should also be an ancestor of x. This is obvious when y is not the parent of x. But even when y is the parent of x, this statement is true because in that case either the left or right child of y has to be x and every node is an ancestor of itself. Hence it is clear that the left or right child of y has to be an ancestor of x.

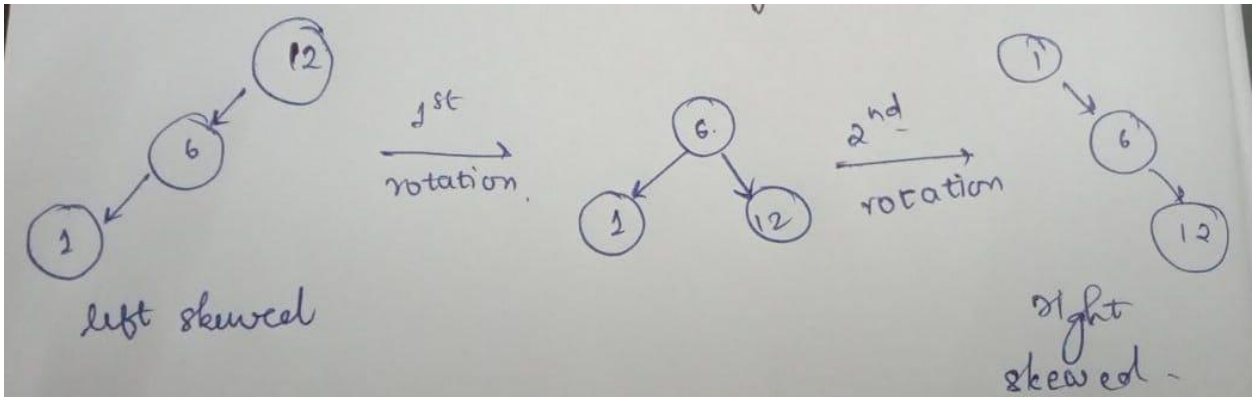
Now assume that the right child of y is an ancestor of x. This means that x belongs to the right subtree of y and x>y. This contradicts the fact that y is the successor of x.Hence the right child of y cannot be an ancestor of x and therefore the left child of y has to be an ancestor of x.

Finally assume that y is not the lowest ancestor of x whose left child is also an ancestor of x. Then let z be the lowest ancestor of x whose left child is also an ancestor of x. Then clearly z has to belong to the left subtree of y and so z<y. But then y cannot be the successor of x because x<z<y. Thus from contradiction, y has to be the lowest ancestor of x whose left child is also an ancestor of x.

Question 4 :

For this, we need to prove that any left skewed binary tree can be converted to right skewed in n-1 rotations.

Example :



Proof :

By definition, each right rotation will increase the length of the rightmost path by at least 1. Therefore, starting from the rightmost path with length 1 (worst case), we need utmost (n-1) rotations to make left skewed to right skewed.

Since the conversion from Left skewed to right skewed involves a maximum number of rotations, any arbitrary shape of binary tree can be rotated to right skewed by a maximum of (n-1) rotations.

By the same argument, the newly right skewed binary tree can be converted to any binary tree by a maximum of (n-1) rotations by doing inverse rotations.

Thus a maximum of 2n - 2 steps are involved, which is O(n).

Therefore, any n-node binary tree can be converted to another n-node binary tree in O(n) time.

Question 5 :

We have to Implement the operations INSERT, DELETE, and SEARCH using singly linked, circular lists.

Time Complexity: Insert Operation: Time complexity of the INSERT function is O(1) as the new node is inserted to the head of the linked list. Delete: Since we are running the loop from start to end to find the required node to delete, the function time complexity is O(n). Search: Again, as we are running the loop from start to end to find the required node, the function time complexity is O(n)

Code:

```
# Defining the Node class which holds the data and the next
Node address
class Node:
    def __init__(self, val):
        self.value = val
        self.next = None
# this is the insert function
# it takes in the head of the linked list and data in the
node to be inserted
# it linked list is empty, ie the head is None, head is made
a node and head next is made to head itself
# otherwise, a new node ‘a’ is created and inserted to the
starting of the list
# the loop link is now attached to 'a'
# also the head of the list is now 'a'
    def INSERT(head, data):
        if head == None:
```

```
            head = Node(data)
            head.next = head
            return head
        a = Node(data)
        a.next = head
        n = head
        while n.next is not head:
            n = n.next
        n.next = a
        head = a
        return head
# this is the delete function
# there are 4 cases we need to take care of
# if linked list is empty it prints "Linked List empty"
# if there is only one element and that element is to be
deleted, head will be made None
# if the element to be deleted is found at head, head has to
be changed to the next value
# all other cases (element to be deleted found at node other
than head)
# if there is no node with value given, "Value not found"
will be printed
    def DELETE(head, data):
        if head == None:
            print("Linked List empty")
            return head
        if head.next is head and head.value == data:
            head = None
            return head
        if head.value == data:
            n = head
            while n.next is not head:
                n = n.next
            n.next = head.next
            head = head.next
            return head
        n = head
        f = head.next
```

```
        flag = 0
        while f.next is not head:
            if f.value == data:
                flag = 1
                break
            f = f.next
            n = n.next
        if flag == 1:
            n.next = f.next
            return head
        else:
            print("Value not found")
            return head
# this is the search function
# if linked list is empty it prints "Value not found"
# if given value is found at head it prints "Value found at
position 1 "
# else it starts from head.next and searches till it reaches
back to the head
# if the given value is found, the function return the
positon in which it is found
# to keep track of the position variable 'count' is used
# if value is not found in the list "Value not found" is
printed
def find(head, data):
        n = head
        count = 1
        if n==None:
            print("Value not found")
            return
        if n.value == data:
            print("Value found at position", count)
            return
        n = n.next
        count = count + 1
        while n != head and n.value != data:
            n = n.next
            count = count + 1
```

```
        if n==head:
            print("Value not found")
            return
        else :
            print("Value found at position", count)
            return
# display function prints the linked list
def DISPLAY(head):
        if head == None:
            print("Linked List empty")
            return
        n = head
        while True:
            print(n.value, end = ' ')
            n = n.next
            if(n==head):
                print('')
                break
        return
# initialising a head for the circular linked list
head = None
# Notice that INSERT and DELETE operations change the linked
list and because of that 'head' is reassigned to what is
returned from the functions
head = INSERT(head, 6);
DISPLAY(head)
head = INSERT(head, 5);
DISPLAY(head)
head = INSERT(head, 4);
DISPLAY(head)
head = INSERT(head, 10);
DISPLAY(head)
head = INSERT(head, 3);
DISPLAY(head)
head = INSERT(head, 2);
DISPLAY(head)
head = INSERT(head, 1);
DISPLAY(head)
```

```
head = DELETE(head, 10);
DISPLAY(head)
head = DELETE(head, 1);
DISPLAY(head)
# Searching for node 2 in the list
find(head ,2);
# Searching for node 7 in the list
find(head, 6);
```

Output :

6
5 6
4 5 6
10 4 5 6
3 10 4 5 6
2 3 10 4 5 6
1 2 3 10 4 5 6 (Inserting in to the linked list)
1 2 3 4 5 6 (after deleting 10)
2 3 4 5 6 (after deleting 1, ie, the head)
Value found at position 1 (search for 2)
Value found at position 5 (search for 6)

Time Complexity:

Insert Operation:

Time complexity of the INSERT function is O(1) as the new node is inserted to the head of the linked list.

Delete:

Since we are running the loop from start to end to find the required node to delete, the function time complexity is O(n).

Search:

Again, as we are running the loop from start to end to find the required node, the function time complexity is O(n)