EE4371 - Assignment 4

Om Shri Prasath, EE17B113

Q1. Show how to implement a stack using two queues. Analyze the running time of the stack operations.

Solution :

Queue is a datatype into which multiple elements can be pushed and also can be popped. It follows a **FIFO** style, meaning that the element first pushed into it, will be the first element to pop out.

Queue is a datatype into which also multiple elements can be pushed and also can be popped. It follows a **LIFO** style, meaning that the element last pushed into it, will be the first element to pop out.

The operations in stack are as follows :

1. `push` : $O(n)$ **Time Complexity**

To implement the push operation of stack which pushes a new element into the stack, we do the following :

1. We first pop out the elements of stack from the queue which is used to store the elements of the stack into another temporary queue using **FIFO** style
2. We then first add the element to be pushed into the original queue, and then pop the elements from the temporary queue and push it into the original queue. Now the original queue which represents the stack contains the element in the correct position

The time complexity of step 1 is $O(n)$, since the time taken to pop from queue and push into queue is $O(1)$ and since we do it n times, it becomes of order $O(n)$. The time complexity of second step is also of order $O(n)$ for the same reason. Thus the overall time complexity is $O(n)$.

1. `pop` : $O(1)$ **Time Complexity**

To implement the pop operation which pops the last inserted element from the stack, we just pop the first element from the queue used to represent the stack, since due to the method of how we implemented the push operation, the first element in the queue will be the last element pushed into the stack, thus popping the first element of queue is equivalent to popping the last inserted element of the stack. Since pop operation in queue is of order $O(1)$, it is the same for this pop operation too.

1. `top` : $O(1)$ **Time Complexity**

To implement the top operation which returns the element at the top of the stack, we can just return the element at the front of the queue which can be accessed using the top operation of the queue. This is of order $O(1)$ too.

1. `length` : $O(1)$ **Time Complexity**

To implement the length operation which returns the length of the stack, we keep a variable which stores the length of the stack. We increment it when we push an element into the stack and decrement if we pop an element.

The edge case of `pop` or `top` from an empty stack can be handled in many ways, here we just return `None` if there is no element in the stack

In [1]:

```python
# Importing queue library

from collections import deque as Queue

# popleft() - Used to pop from queue
# append() - Used to push into queue


"""
Implementation of Stack
"""


class Stack:

    """
    Constructor method
    """

    def __init__(self):

        # Queue to hold elements of stack
        self.q_stack = Queue()

        # Queue to hold elements temporarily
        self.q_temp = Queue()

        # Variable to hold the stack length
        self.l = 0

    """
    Method to push element into stack
    """

    def push(self, elem):

        # Pop all elements from main queue into temp queue
        while self.q_stack:
            self.q_temp.append(self.q_stack.popleft())

        # Push the new element into main queue
        self.q_stack.append(elem)

        # Pop all elements from temp queue into main queue
        while self.q_temp:
            self.q_stack.append(self.q_temp.popleft())

        # Increase length of stack
        self.l += 1

    """
    Method to pop element from stack
    """

    def pop(self):

        # Check if stack is empty
        if self.l == 0:
            return None

        # Decrease the length of the stack
        self.l -= 1

        # Return the popped element
        return self.q_stack.popleft()

    """
    Method to get the top element of stack
    """

    def top(self):

        # Check if stack is empty
        if self.l == 0:
            return None

        # Return the top element
        return self.q_stack.top()

    """
    Method to get the length of stack
    """

    def length(self):

        # Return the length
        return self.l

    """
    Method to print the stack
    """

    def display(self):

        # Print the stack

        print(self.q_stack)


# Testing the implementation

s = Stack()

s.push(3)
print("Pushed 3 into the stack")
s.push(2)
print("Pushed 2 into the stack")
s.push(6)
print("Pushed 6 into the stack")
s.display()
print("Popped", s.pop(), "from the stack")
print("Popped", s.pop(), "from the stack")
s.display()
print("Length of Array =", s.length())
s.push(4)
print("Pushed 4 into the stack")
s.push(10)
print("Pushed 10 into the stack")
s.push(15)
print("Pushed 15 into the stack")
s.display()
print("Popped", s.pop(), "from the stack")
print("Popped", s.pop(), "from the stack")
s.display()
print("Length of Array =", s.length())
```

```
Pushed 3 into the stack
Pushed 2 into the stack
Pushed 6 into the stack
deque([6, 2, 3])
Popped 6 from the stack
Popped 2 from the stack
deque([3])
Length of Array = 1
Pushed 4 into the stack
Pushed 10 into the stack
Pushed 15 into the stack
deque([15, 10, 4, 3])
Popped 15 from the stack
Popped 10 from the stack
deque([4, 3])
Length of Array = 2
```

Q2. Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

Solution :

The hash index, which is the position where the key is to be stored in the hash table, is decided using the hash function $k \bmod 9$, which means there are 9 slots to store the keys.

The collisions, which means that more than one key has the same hash index, is resolved by chaining where we create a chain of keys which is created mostly as a singly linked list. Thus, we store a chain of keys in the hash table.

The steps of creating the hash table are as follows :

Initial Hash Table (Empty) :

Hash Index	Keys
0	
1	
2	
3	
4	
5	
6	
7	
8	

Add key $= 5$**, Index** $= 5 \bmod 9 = 5$

Hash Index	Keys
0	
1	
2	
3	
4	
5	5
6	
7	
8	

Add key $= 28$**, Index** $= 28 \bmod 9 = 1$

Hash Index	Keys
0	
1	28
2	
3	
4	
5	5
6	
7	
8	

Add key $= 19$**, Index** $= 19 \bmod 9 = 1$

Hash Index	Keys
0	
1	28→19
2	
3	
4	
5	5
6	
7	
8	

Add key $= 15$**, Index** $= 15 \bmod 9 = 6$

Hash Index	Keys
0	
1	28→19
2	
3	
4	
5	5
6	15
7	
8	

Add key $= 20$**, Index** $= 20 \bmod 9 = 2$

Hash Index	Keys
0	
1	28→19
2	20
3	
4	
5	5
6	15
7	
8	

Add key $= 33$**, Index** $= 33 \bmod 9 = 6$

Hash Index	Keys
0	
1	28→19
2	20
3	
4	
5	5
6	15→33
7	
8	

Add key $= 12$**, Index** $= 12 \bmod 9 = 3$

Hash Index	Keys
0	
1	28→19
2	20
3	12
4	
5	5
6	15→33
7	
8	

Add key $= 17$**, Index** $= 17 \bmod 9 = 8$

Hash Index	Keys
0	
1	28→19
2	20
3	12
4	
5	5
6	15→33
7	
8	17

Add key $= 10$**, Index** $= 10 \bmod 9 = 1$

Hash Index	Keys
0	
1	28→19→10
2	20
3	12
4	
5	5
6	15→33
7	
8	17

Q3. Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Solution :

To prove the above, we need to use the property of BST that the elements in the left subtree of a node is lesser than the node and the elements in the right subtree of a node is greater then the node.

We have to prove three statements to prove that y is the lowest ancestor of x whose left child is also an ancestor of x

1. y is an ancestor of x
2. The left child of y is an ancestor of x
3. y is the lowest ancestor which satisfies the above conditions

Claim 1: y **is an ancestor of** x

Let us assume that y is not an ancestor of x. This means there exists a w which is the first common ancestor of x and y. From the property of BST, we have $x < w < y$, but this means that y cannot be the successor of x. Thus our assumption was wrong. Hence y is an ancestor of x.

Claim 2: The left child of y **is an ancestor of** x

Let us assume that the left child of y is not an ancestor of x. But since we proved above that y is an ancestor of x, this must mean that the right child of y is an ancestor of x, i.e. x belongs in the right subtree of y. But that implies that $x > y$, which goes against the notion that y is the successor of x. Thus our assumption was wrong. Hence the left child of y is an ancestor of x

Claim 3: y **is the lowest ancestor which satifies the above condition**

Let us assume that there exists an w which satisfies the condition that w is an ancestor of x and the left child of w is an ancestor of x. But since y is also an ancestor of x, we have $x < w < y$, which means w will be the successor of x. Thus, it means only the successor of x can satisfy the above conditions.

Therefore, if y is the successor of x, then y must be the lowest ancestor of x whose left child is also an ancestor of x.

Q4. Show that any n-node binary tree can be converted to any other n-node binary tree using $O(n)$ rotations.

Solution :

For converting an n-node binary tree into another n-node binary tree, we take an intermediate step of converting the initial n-node binary tree into an right-going tree using $O(n)$ rotations and convert the right going tree into the final form using $O(n)$ rotations, thus overall taking $O(n)$ rotations.

We need to prove that an arbitrary n-node binary tree can be converted into a right-going tree in $O(n)$ rotations. The algorithm to convert the tree is as follows:

1. Take a left child of any element on the right subtree of the binary tree
2. We apply a right rotation to this node, which increases the number of elements in the right-most tree by 1 due to the property of right rotation.
3. We repeat this until the tree becomes right-going.

We see that one step of the above algorithm increases the number of elements in the right subtree by 1. Thus for a tree with k elements in left-subtree, we need k steps of the above algorithm to convert the tree into a right going tree.

The worst case situation occurs when the initial tree of n nodes is left-going tree, i.e. te left most chain has $n-1$ elements. To change this tree into right-going tree, we need to do $n-1$ rotations to bring all the elements to the right. Thus the required number of rotations to convert a arbitrary n node tree to a right going tree is $O(n)$.

Since the number of rotations required to convert an arbitrary n node tree to a right going tree is $O(n)$, the reverse of converting a right going tree to an arbitrary n-node tree also requires $O(n)$ rotations since we only need to apply the rotations in reverse to get the tree.

Thus, from the above statements, we have shown that converting any n-node binary tree to any other n-node binary tree takes $O(n) + O(n) = O(n)$ rotations

Q5. Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

Solution :

1. `INSERT` : $O(1)$ **Time Complexity**

For inserting a new node, we just reconnect the current head to point to this new node and the new node to point to the node which the head node was initially pointing too. This takes $O(1)$ time complexity

1. `DELETE` : $O(n)$ **Time Complexity**

For deleting a node, we need to search if the node is present in the circular linked list and if it is present we remove it by connecting the node previous to the node to be deleted to the node next to the node to be deleted, which effectively removes the node from the chain. To find the element the worst case time is to traverse the full circular linked list, thus the time complexity is $O(n)$.

1. `SEARCH` : $O(n)$ **Time Complexity**

For searching a node as stated above the worst case time complexity involves searching the whole linked list, thus the time complexity is $O(n)$

In [2]:

```python
"""
Python code to implement dictionary operations
"""


"""
Implementation of nodes of circular linked list
"""
class Node:
    def __init__(self, val):

        self.val = val
        self.next = None

"""
Implementation of circular linked list wih dictionary options
"""
class CircularLinkedList:
    def __init__(self):

        self.head = None

    """
    INSERT operation
    """
    def INSERT(self, node):

        # Checking if head is defined
        if self.head:
            # Inserting the node as the node next to head
            node.next = self.head.next
            self.head.next = node

        else:
            # Initializing node as head if head is not present
            self.head = node
            # Linking the head to itself
            self.head.next = self.head

    """
    DELETE operation
    """
    def DELETE(self, node):

        # Checking if head is defined
        if self.head:
            prev = self.head
            # Loop to check if node is present
            while(True):
                # Element present
                if(prev.next==node):
                    # Removing element from linked list
                    if(prev.next == prev):
                        prev = None
                    else:
                        if(self.head==node):
                            self.head = prev
                        prev.next = prev.next.next
                    print("Deleted Element")
                    return
                prev = prev.next
                # Coming back to head -> Element not found
                if(prev==self.head):
                    print("Element not found")
                    return

        else:
            # No elements present
            print("No elements to delete!")

    """
    SEARCH operation
    """
    def SEARCH(self, val):

        # Checking if head is present
        if self.head:
            node = self.head
            # Loop to check if node is present
            while(True):
                if(node.val==val):
                    print("Element Found")
                    return
                node = node.next
                # Coming back to head -> Element not found
                if(node == self.head):
                    print("Element not Found")
                    return
        else:
            # No elements present
            print("Empty Circular Linked List")

    """
    Function to fancy print linked list
    """
    def PRINT(self):

        t = self.head

        arr = []

        while(t.next!=self.head):
            arr.append(t.val)
            t = t.next

        arr.append(t.val)

        t = " "

        for i in arr:
            t = t+("> "+str(i)+" ").rjust(10,"-")

        t+=" ".ljust(8,"-")+""
        print(t)
        print("|"+" "*(len(t)-1)+"|")
        print(" "+"-"*(len(t)-1))

# Testing Code

from random import sample

circ_list = CircularLinkedList()

l = []

r_sample = sample(range(50),10)

for i in r_sample:
    l.append(Node(i))

for i in l[:-1]:
    print("Insering",i.val)
    circ_list.INSERT(i)

circ_list.PRINT()

print("\nSearching for",l[0].val)
circ_list.SEARCH(l[0].val)
print("\nSearching for",l[-1].val)
circ_list.SEARCH(l[-1].val)
print("\nSearching for",l[3].val)
circ_list.SEARCH(l[3].val)

print("\nDeleting",l[3].val)
circ_list.DELETE(l[3])
circ_list.PRINT()

print("\nDeleting",l[0].val)
circ_list.DELETE(l[0])
circ_list.PRINT()
```

```
Insering 46
Insering 22
Insering 30
Insering 36
Insering 3
Insering 5
Insering 39
Insering 23
Insering 45
 -----> 46 -----> 45 -----> 23 -----> 39 ------> 5 ------> 3 -----> 36 -----> 30 -----> 22  -------
|                                                                                                 |
 -------------------------------------------------------------------------------------------------

Searching for 46
Element Found

Searching for 25
Element not Found

Searching for 36
Element Found

Deleting 36
Deleted Element
 -----> 46 -----> 45 -----> 23 -----> 39 ------> 5 ------> 3 -----> 30 -----> 22  -------
|                                                                                        |
 ----------------------------------------------------------------------------------------

Deleting 46
Deleted Element
 -----> 22 -----> 45 -----> 23 -----> 39 ------> 5 ------> 3 -----> 30  -------
|                                                                              |
 ------------------------------------------------------------------------------
```