Assignment Number [4]: Algorithms,2020

Nikilesh B
EE17B112

1. A stack can be implemented using two queues. Let the stack to be implemented be ‘s’ and queues used to implement be ‘q1’ and ‘q2’.

Stack ‘s’ can be implemented in two ways:

Method 1 (By making push operation costly)

This method makes sure that a newly entered element is always at the front of ‘q1’, so that pop operation just dequeues from ‘q1’. ‘q2’ is used to put every new element in front of ‘q1’.

1. **push(s, x)** operation’s step are described below:
    - Enqueue x to q2
    - One by one dequeue everything from q1 and enqueue to q2.
    - Swap the names of q1 and q2
2. **pop(s)** operation’s function are described below:
    - Dequeue an item from q1 and return it.

Method 2 (By making pop operation costly)

In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

1. **push(s, x)** operation:
    - Enqueue x to q1 (assuming size of q1 is unlimited).
2. **pop(s)** operation:
    - One by one dequeue everything except the last element from q1 and enqueue to q2.
    - Dequeue the last item of q1, the dequeued item is the result, store it.
    - Swap the names of q1 and q2
    - Return the item stored in step 2.

Python implementation of the first method is given below:

```
# Program to implement a stack using
# two queue
from queue import Queue
class Stack:

    def __init__(self):
        self.q1 = Queue()         # Two inbuilt queues
        self.q2 = Queue()
        self.curr_size = 0

    def push(self, x):
        self.curr_size += 1
        self.q2.put(x) # Push x first in empty q2

        while (not self.q1.empty()): # Push all the remaining
            self.q2.put(self.q1.queue[0]) # elements in q1 to q2.
            self.q1.get()

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

    def pop(self):
        if (self.q1.empty()):         # if no elements are there in q1
            return
        self.q1.get()
        self.curr_size -= 1

    def top(self):
        if (self.q1.empty()):
            return -1
        return self.q1.queue[0]

    def size(self):
        return self.curr_size
```

```
Sample Input testing:
    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())
```

```
Output :
current size: 3
3
2
1
current size: 1
```

2.We insert the keys 5,28,19,15,20,33,12,17,10 into a hash table with collisions resolved by chaining.

The table have 9 slots, and the hash function is h(k)=k mod 9.

Let us number our slots 0,1,…,8.

Here we have:

0. 5 mod 9 → 5
1. 28 mod 9 → 1
2. 19 mod 9 → 1
3. 15 mod 9 → 6
4. 20 mod 9 → 2
5. 33 mod 9 → 6
6. 12 mod 9 → 3
7. 17 mod 9 → 8
8. 10 mod 9 → 1

As we can see, collisions happened for **h(28) = h(19) = h(10) = 1** and **h(15) = h(33) = 6.** Wherever collisions occurred are resolved by chaining.

Then our resulting hash table will look like :

h(k)	keys
0 mod 9	
1 mod 9	10 → 19 → 28
2 mod 9	20
3 mod 9	12
4 mod 9	
5 mod 9	5
6 mod 9	33 → 15
7 mod 9	
8 mod 9	17

3.

- First we establish that y must be an ancestor of x.
- If y weren't an ancestor of x, then let z denote the first common ancestor of xxx and yyy.
- By the binary-search-tree property, x<z<y , so y cannot be the successor of x..
- Next observe that y.left must be an ancestor of x because if it weren't, then y.right would be an ancestor of x, implying that x>y.
- Finally, suppose that y is not the lowest ancestor of x whose left child is also an ancestor of x.
- Let z denote this lowest ancestor. Then z must be in the left subtree of y, which implies z<y, contradicting the fact that y is the successor of x.

Thus , it is shown that y is the lowest ancestor of x whose left child is also an ancestor of x .

4.

- Basically, you need to fill the stack up with the n numbers starting from 0. then pop them all to get your first permutation.
- To get the second possible permutation you need to do the same thing but this time start from 1 to n and your last item will be the one at position 0.
- You need to do it all the way to the n and then you have to do it the other way around, starting from n to 0 and then n-1 to 0 with the last item being n.

To illustrate this let's consider the set of 3 integers {1,2,3}.

We will use **( )** to show stack and **{ }** to show the resulting permutation after popping all the elements of the stack.

(1|2|3) pop {3,2,1}

(2|3|1) pop {1,3,2}

(3|1|2) pop {2,1,3}

now the other way around

(3|2|1) pop {1,2,3}

(2|1|3) pop {3,1,2}

(1|3|2) pop {2,3,1}

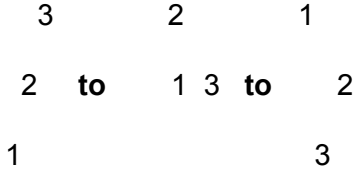
5. Let

LEFT = an entire left linked list binary tree

RIGHT = an entire right linked list binary tree.

You can easily rotate LEFT to RIGHT in (n-1) rotations

e.g: n = 3



Since 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), you need at most (n-1) rotations performed to make it into RIGHT.

Thus, you can easily conclude that any arbitrary shape of binary tree with n nodes can rotate into RIGHT within (n-1) rotations. Let T_1 be the node you begin with Let T_2 be the node you end with.

You can rotate T_1 to RIGHT within (n-1) rotations. Similarly, You can rotate T_2 to RIGHT within (n-1) rotations.

Therefore, To rotate T_1 into T_2, simply rotate T_1 into RIGHT , then do the inverse rotation to rotate from RIGHT into T_2.

Therefore, you can do this in (n-1)+(n-1) = 2n-2 rotations in the upper bound, which is nothing but O(n) rotations.