COMP 6481

Assignment 2: Theory Part

Amro Elbahrawy

40221760

Question 1:

a)

My idea is for the array to have two pointers, one at the beginning and one at the end, where each pointer will correspond to each stack pointer.

Assumption: Since the array size is fixed and cannot be changed, I will assume that the both stacks are identical in sizes.

Pushing or popping elements will either increment or decrement the pointers depending on which stack it is.

If the pointers are equal to the end or beginning of the array, that means that the stack is empty. Size will be equal to the current index of the pointers. (index for pointer 1, and array size – index for pointer 2)

b)

array[] = new array[N]

pointer_stack1 = 0

pointer_stack2 = N

////// **Size methods**

int size1(){
return pointer_stack1;
}
	Int Size2(){
	Return pointer_stack2-N
	}

///// **isEmpty methods**

```
Boolean isEmpty1(){
if(pointer_stack1 == 0)
        then true
else false
}
```

```
Boolean isEmpty2(){
if(pointer_stack2 == N)
        then true
else false
}
```

///// **isFull methods**

```
boolean isEmpty1(){
if(pointer_stack1 == (N/2)-1)
        then true
else false
}
```

```
boolean isEmpty2(){
if(pointer_stack2 == (N/2))
        then true
else false
}
```

/////// **Push methods**

```
push1(e){
if(pointer_stack1 > (N/2)-1) ///Stack 1 is full
        print("Stack 1 is full, cannot insert")
else
      array[pointer_stack1] = e
       pointer_stack1++
}
```

```
push2(e){
if(pointer_stack2 < (N/2))
        print("stack 2 is full, cannot insert")
else
      array[pointer_stack2-1] = e
       pointer_stack--

}
```

/////**Pop methods**

```
pop1(){
if(pointer_stack1 == 0)
```

```
pop2(){
if(pointer_stack2 == N)
```

```
    print("stack 1 is empty")
    return null

else{
    elem = array[pointer_stack1-1]
     pointer_stack1  = pointer_stack1 -1
      return element
}

}
```

```
    print("stack 2 is empty")
    return null

else{
    elem = array[pointer_stack2]
     pointer_stack2  = pointer_stack2 +1
      return element
}

}
```

c)

Size Methods: O(1). Because the sizes is just the array size – current stack pointers.

isFull/isEmpty Methods: O(1). Since we just check the stack pointer variables without any looping/traversal.

Push methods: O(1). It's just an array assignment of 1 index and changing the pointer value without any loops.

Pop Methods: O(1). Same logic for Push methods, but with an additional variable to store the value of the popped value to return it.

d) The Ω for all methods will be Ω(1) for the same reason why O was O(1) for all of them, since it's just assignments of 3 variables at maximum per method without any loops.

For 3 stacks, I think it would be more complicated in the case where the array size is not divisible by 3 because some stacks will be larger than others, and that needs to be accounted for.

But in the case where N divides 3, a 3rd pointer and boundary variables for each stack should solve the 3 stacks problem in the same manner that it was solved for 2 stacks.

Question 2:

Input: Stack T

Output: all possible subsets of T

```
Stack S1
Queue Q1
Print(empty.pop())
For(i = 1 to T.size)
{
    For(j = 1 to i)
    {
        Temp = T.pop()
        S1.push(temp)
        Print(temp)
        Q1.enqueue(temp)
    }
    While(!Q1.isempty){
        S1.push(Q1.dequeue)
    }

}
```

a) **Time Complexity:** Since we have two nested loops inside 1 loop, they both will run "n" times in the worst case. Since these nested loops are not nested within each other, both of their complexity will be equivalent to O(n). Therefore, the time complexity of this algorithm is O(n^2)

b) **Space Complexity:** We are using 2 stacks and 1 queue to move around the values in order to print them, reaching to n^3 in the worst case, so the space complexity in the worst case should be O(n^3)

Question 3:

a)

Input: T1, T2

Output: Boolean value (True or False)

```
If(T1.root .compareTo(T2.root) == 0)
        Return true;
Recursion(T1.left, T2.left)
Recursion(T1.right, T2.right)

Return false
```

b) Since both trees vary in sizes, and we will traverse all of T2, then that results in O(n) (Assuming the number of nodes in T2 is "m"). In the worst case, we also might have to traverse all of T1 as well. Assuming that the number of nodes in T1 is "m", then the time complexity should be O(m*n)

c)I think the algorithm would still succeed if duplicate values were allowed, because the comparison method that is used is compareTo(), which returns a value indicating if two nodes are equal. So it shouldn't matter if a node is duplicated or not because we are comparing in identical relative positions of each tree.

Question 4:

Assuming the last W in the preorder sequence is V, here is a tree that satisfies both: