CS583 Assignment 3 Mithilaesh Jayakumar(G01206238)

1. Describe how to implement a queue using two stacks and O(1) additional memory, so that the amortized time for any enqueue or dequeue operation is O(1). The only access you have to the stacks is through the standard subroutines Push and Pop.

A stack works based on the Last In First Out (LIFO) concept where the last pushed element is popped out first. Now we have a second stack which is going to store the first stack elements in reverse order. We put our populated stack to the left and our new empty stack to the right. To reverse the order of the elements in the first stack, we are going to pop each element from the left stack, and push them to the right stack.

In order to construct a queue model we will consider these two stacks. One stack will be used for enqueue operation (stack #1 on the left, will be called as Input Stack), another stack will be used for the dequeue operation (stack #2 on the right, will be called as Output Stack).

The enqueue and dequeue operations of this queue model works as follows.

Enqueue:

- Push the new element into Input Stack

Dequeue:

- If Output Stack is empty, refill it by popping each element from Input Stack and pushing it onto Output Stack
- Pop and return the top element from Output Stack

Using this method, the Time complexity is O(1) for Enqueue operation, since appending an element to the stack is a constant time operation and the Time complexity for Dequeue operation is O(1) amortized and O(n) worst-case.

In the worst case scenario when the Output stack is empty, the algorithm pops n elements from the Input stack and pushes n elements to the Output Stack, where n is the queue size. Since we have n pop operations, using the worst-case per operation analysis gives us a total of O(n^2) time. The number of times a pop operation can be called is limited by the number of push operations before it. Although a single pop operation could be expensive, it is expensive only once per n times, when the Output stack is empty and when there is a need for data transfer between Input and Output stack.

Hence the total time complexity of the sequence is : n (for push operations) + 2*n (for first pop operation) + n - 1 (for pop operations) which is O(2*n).This gives O(2n/2n) = O(1) average time per operation. Therefore the overall Time complexity is O(m) for m operations, i.e. O(1) average time per operation, which can be seen from the fact that each element is pushed no more than twice and popped no more than twice.

2. An ordered stack is a data structure that stores a sequence of items and supports the following operations.
OrderedPush(x): removes all items smaller than x from the beginning of the sequence and then adds x to the beginning of the sequence.
Pop: deletes and returns the first item in the sequence (or Null if the sequence is empty).
Suppose we implement an ordered stack with a simple linked list, using the obvious OrderedPush and Pop algorithms. Prove that if we start with an empty data structure, the amortized cost of each OrderedPush or Pop operation is O(1).

Amortized analysis using the aggregate analysis is defined to be the average cost. In the aggregate analysis, a sequence of n operations takes the worst-case time T(n) in total. In the worst case, the average cost per each operation is T(n)/ n.

From the example, the elements of the stack are as follows:

The top of the stack is 3.

First we will show this for pop. According to the specification, the operation deletes the smaller elements less than the top of the stack value every time by returning the first item in the stack. The result for our given stack is as follows:

1. 1 <3. So, 1 popped out.
2. 2 <3. So, 2 popped out.
3. 4 is not less than 3. SO; 4 is not popped.
4. 5 is not less than 3. So, 5 is not popped.

5. 6 is not less than 3. So, 6 is not popped.

Since we just remove the head of the linked list, this takes constant time. It is independent of the size of the stack, so it is O(1) time worst case, so obviously also O(1) amortized.

According to the specification, the orderedPush (x) operation removes all the items smaller than x from the top of the stack. It then adds x to the top of the stack. It uses peek ( ) operation on the stack, which returns the stack top element. The function checks whether the stack is empty and peek ( ) is less than x using while loop. If not, it pops the element from the stack only when the 2 conditions are satisfied.

For example if we have k items on the stack, and our item x being pushed on is larger than all items, we will have to check and remove all items, so it will take O(k) time.

However every item can be removed at most once. So using a charging scheme, for every push charge the item x with 3 operations. The first 2 of these operations are used when we check against the top of the stack and find that it is larger than x, so we then add it to the stack. We know that whenever we do a push, we will only find one item larger than x (since then we are done) and we only add x once as well, at the end.

The last of the 3 operations is used when an item is compared to x when it is in the stack and x is removed. Whenever an item is being compared to x we know one of two things will happen. Either the item is smaller than x, so it is part of the initial 2 costs charged to it, or the item is larger. In the later case we use the last operation charged to x to remove it from the list. Since it is removed we will no longer have to compare anything to x.

Since every operation in the push is accounted for, we know the total time for n pushes is at most 3n, which is O(n). So the amortized cost of a push is O(1). The result on our stack will be as follows:

1. For ordered Push(6) , 6 < 6. No elements are less than 6 Here, peek ( )=6
2. For ordered Push(5) , 6 > 5. So, no elements are popped out Here, peek ( ) =6.
3. For ordered Push(4), 5 < 4. So, no elements are popped out Here, peek( ) =5.
4. For ordered Push(2) , 4 > 2 So, no elements are popped out Here, peek ( ) =4.
5. For ordered Push (1) , 3 > 1. So, no elements are popped out Here, peek ( ) =3.
6. For ordered Push(3) , 1 < 3 Here, peek ( ) =1. So, the elements less than 3 are popped out and 3 occupies the top of the stack position.

We can also incorporate the pop into this charging scheme as well. When we pop off the top item, you can consider that as using the last operation charged to the item. So any sequence of n pushes and pops will take O(n) total time, or O(1) amortized time per operation.