ESO207: Data Structures and Algorithms
Abhimanyu Sethia
Roll Number: 190023
Section: S9

Assignment 2

Abhimanyu Sethia

Q1 In lectures we saw implementation of a queue using an array. Suppose instead of an array you are given two empty stacks, S_1, S_2. You may operate on S_1, S_2 using stack operations only, you have no access to implementation of S_1, S_2.
***(a)** Describe a strategy to design a queue using S1, S2 and some integer variables.*

Problem: A stack is a LIFO (Last In First Out) data structure, i.e. if we pop an element from a stack, then the last (latest to be inserted) element gets deleted. However, a queue is a FIFO (First In First Out) data structure, i.e. if we dequeue an element from a queue, then the first (earliest to be inserted) element is deleted.

Idea: Hence to implement a dequeue operation with the help of stacks, we would need to invert the order of elements in the stack and then pop it.

Solution: We will store the elements in the order in which they have been inserted, in one of the stacks, say S_1. While in the other stack S_2, we will store the elements in the inverse order of that they have been inserted.
Hence, to implement enqueue operation, we will simply push the element into S_1 and to implement dequeue operation, we will pop an element from S_2. If S_2 is empty, then we will first invert elements (by popping elements from S_1 and pushing them to S_2) and then pop from S_2, to dequeue.
This will preserve the FIFO property of the queue.

Objectives of Operations

1. createQueue(Q)- creates two stacks
2. isEmptyQueue(Q)- returns True if the queue is empty; else returns False
3. isFullQueue(Q)- returns True if the queue is full; else returns False
4. Enqueue(Q,x)- inserts element x in the queue i.e. pushes element x in stack S_1 (as explained above)
   prints 'Queue is full' if Queue is Full.
5. Dequeue(Q)-deletes the first-in element and returns it
   i.e. if S_2 is empty, first inverts the stack, by popping elements from S_1 and then pushing them in S_2. If not, then this step is skipped. After populating S_2, it pops an element from S_2 and returns it.

(b) *Write pseudo-code for queue operations using your strategy. What is the complexity of various queue operations, in terms of the number of elements in the queue?*

The pseudocode assumes the following stack operations are available (as these were shown in the Lecture)-

1. createStack(S)- initiates an instance of the stack data structure
2. isEmptyStack(S)- returns True if stack is empty and False if not
3. isFullStack(S)- returns True if stack is full and False if not
4. push(S,x)- pushes the element x into stack S
5. pop(S)- deletes and returns the last/latest-inserted element of the stack (LIFO)

Pseudocode

```
createQueue(Q)
    createStack(S1)
    createStack(S2)

isEmptyQueue(Q)
    if isEmptyStack(S1)==True and isEmptyStack(S2)==True
    then return True
    else return False

isFullQueue(Q)
    if isFullStack(S1)
    then return True
    else return False

Enqueue(Q,x)
    if isFullQueue(Q)==True
    then print 'Queue is Full'
         return
    push(S1,x)

Dequeue(Q)
    if isEmptyQueue(Q)
    then print 'Queue is empty'
         return
    if isEmptyStack(S2)
    then while isEmptyStack(S1)==False
         do temp=pop(S1)
            push(S2, temp)
         //endWhile
    //endIf
    return pop(S2)
```

Time Complexity

- We know that all the stack operations are O(1). Since createQueue(Q), isEmptyQueue(Q), isFullQueue(Q) and Enqueue(Q,x) simply consist of a fixed finite number of stack operations, they will all have time complexity **O(1)**.
- For Dequeue(Q), the worst case will be when S_2 is empty. Hence, if there are n elements in the queue, all of them must be in S_1. So, the while loop will iterate n times, and hence, the worst-case time complexity of Dequeue(Q) is **O(n)**.

(c) *Briefly argue for correctness of your implementation.*

1. createQueue(Q)- clearly creates two stacks, namely S_1 and S_2. This was the objective as well. Hence, it is trivially correct.
2. isEmptyQueue(Q)- There are two exhaustive possibilities-
   (i) The Queue is empty
   $\implies$ S_1 is empty $\implies$ isEmptyStack(S_1) is True
   $\implies$ S_2 is empty $\implies$ isEmptyStack(S_2) is True
   Since the if condition is a logical conjunction of the two and both of them are True, the if condition is rendered True, and thereby, the function returns True. This is the correct output (as per our objective).
   (ii) The Queue is not empty
   $\implies$ at least one of the stacks, S_1 and S_2, will be non-empty $\implies$ for at least one of S_1 or S_2, isEmptyStack(S) will return False.
   Since the if condition is a logical conjunction of the two and at least one of them is False, the if condition is rendered False and thereby, the function returns False. This is the correct output (as per our objective).

   Since correctness of the function in each of the two cases has been established and the two cases are exhaustive, the correctness of the function in all cases follows.
3. isFullQueue(Q)- The statement "A queue is full" is equivalent to saying that "no additional elements may be inserted in the queue." As stated in our strategy, enqueue operation inserts all elements in the stack S_1. Therefore, the queue will not be able to accommodate additional elements if and only if the stack S_1 gets full.
   In other words, if the stack S_1 is full, then the queue is full. Otherwise, the queue is not full.
   This can be trivially seen in the function and hence, its correctness, follows from the correctness of the above statement.
4. Enqueue(Q,x)- We have already defined in our strategy that we will store elements in stack S_1, in the order in which they have been inserted. Therefore, the enqueue function is supposed to push an element to stack S_1 and they will naturally fall in the order in which they are enqueue-ed. As an exception, enqueue function is expected to print 'Queue is Full' and return, if the queue is full.
   Hence, the function's correctness trivially follows from the correctness of the isFullQueue(Q) function and the objective of the function.
5. Dequeue(Q)- Queue is a FIFO (First In First Out) data structure. Hence, to prove the correctness of the dequeue function, we need to prove that it deletes and returns the first-in element of the queue (i.e. earliest element to be inserted of all the elements that are still in the queue).

(a) Case 1- If S_2 is empty,
then the dequeue function pops elements from S_1 and pushes it to S_2, iteratively. Since the elements are stored in S_1, in the order in which they have been inserted, the first-in element of the queue would be the first element of the stack S_1. And hence, in the while loop (line 26-29), it would be the last element to be popped and thereby, the last element to be pushed to S_2. It follows, hence, that the first-in element (earliest to be inserted in the queue) would be the last element of the stack S_2.
Since dequeue returns pop(S_2) and stack is a LIFO data structure, it follows that the first-in element (earliest to be inserted in the queue of all the elements currently in the queue) is deleted and returned.

(b) Case 2-- If S_2 is not empty
When any Queue is initialised, S_2 is empty. Moreover, the only operator which pushes any elements in stack S_2 is Dequeue. Hence, the elements in S_2 have to be the remaining elements, after some previous call of Dequeue.
In case-1,it has already been proved that the Dequeue operation inverts the order of elements i.e. the order of elements in S_2 is inverse of the order in which they have been inserted in the queue.
Hence, the earliest inserted element which is still present in the queue, will be the last (topmost) element in S_2.
Since dequeue returns pop(S_2) and stack is LIFO data structure, it follows that the first-in element (earliest to be inserted in the queue of all the elements currently in the queue) is deleted and returned.

The correctness of dequeue, hence, follows from the correctness of dequeue in each of the above two cases.

Q2 In lecture 9, we saw how to convert recursive procedure for prefix (also called pre-order) traversal of a binary tree to a non-recursive procedure using a stack
(a) *Write pseudo-code for a recursive procedure for inorder traversal of a binary tree.*

```
1  inorderTraversal(T)
2      if T ≠ nil
3      then inorderTraversal(T.lchild)
4           print(T.val)
5           inorderTraversal(T.rchild)
```

(b) *Now convert the recursive procedure in part (a) to an equivalent non-recursive procedure using a stack. Write the pseudo-code of your non-recursive procedure.*

```
1  inorderTraversal(T,S)
2      curr=T
3      while curr≠nil or isEmptyStack(S)==False
4      do  if curr≠nil
5          then push(S,curr)
6               curr=curr.lchild
7               continue
8          if isEmptyStack(S) == False
9          then temp=pop(S)
10              print(temp.val)
```

```
11                curr=temp.rchild
12      //endWhile
```

Note: The code above defines function inorderTraversal(T,S) for the traversal of a single binary tree. In case of multiple trees, we could have another function, which calls the function inorderTraversal(T,S) iteratively for each tree. Correctness of such a function would trivially follow from the correctness of inorderTraversal(T,S)

(c) *Argue for correctness of your code in part (b).*

To prove that the function is correct we need to prove the following-

(i) The function prints every node exactly once

(ii) The left child of a node, if not nil, is printed before the node itself.

(iii) The right child of a node, if not nil, is printed after the node itself.

Let k be the number of layers in the binary tree. We prove the correctness of the code by the principle of **strong** induction on k.

Base Case: If $k = 1$, the tree consists of a single node (let us call the node x). Initially curr will point to x. Since $x \neq$nil, the while guard condition and the first if condition are True. The control will come to line 5, x will be pushed to the stack S and curr will now point to nil (since the lchild of x is nil).
In the next iteration, the second if condition will hold True and temp will point to x (since x will be the popped element from stack). The program will print the value of x and curr will point to nil (as rchild of x is nil).
In the third iteration, the while guard condition will be rendered False (as curr is nil and the stack is empty) and so, the loop will exit.

Therefore, only the value of x will be printed exactly once. Hence, the three conditions (i), (ii) and (iii) are True for the base case.

Induction Hypothesis: Let us assume that the three conditions (i), (ii) and (iii) are true for all binary trees, with $k \leq n$.

Induction Step: We need to show that (i), (ii) and (iii) hold for a tree with $k = n + 1$. Let there be a tree T, with root r and $k = n+1$. Let the two children sub-trees of r be T_l and T_r. Clearly, T_l and T_r are subtrees with $k \leq n$ (and at least one of them must have $k = n$, as T is assumed to have $k = n + 1$). Note that T_l or T_r may be nil as well.
Initially, curr is initialised to r. In the first iteration, r is pushed to the stack and curr points to T_l. Then the following cases arise-

1. Case 1: T_l is nil
   Then, in the next iteration, the first if condition is False and the second if condition is True. So control comes to line 9, and temp is initialised as r (as r will be returned by pop(S)). The value of r is printed and curr is assigned T_r. In the following iterations, the function traverses sub-tree T_r
   So we have-

- Traversal of T_r follows the conditions (i), (ii) and (iii) (from our Induction Hypothesis)
- r is printed only once and before T_r

Hence, it follows that traversal of tree T also satisfies conditions (i), (ii) and (iii) valid.

2. Case 2: T_l is not nil
Then, in the next iteration, T_l is pushed to the stack. The traversal of the subtree T_l will be same as the traversal of any tree with $k \leq n$. So (i), (ii) and (iii) conditions are satisfied for the traversal of T_l (from our induction hypothesis).
After traversal of T_l, curr will point to nil and only r will be left in the stack. In the next iteration, the second if condition (line 8) will be true and so temp will point to r (line 9), value of r will be printed (line 10) and curr will now point to T_r (line 11). Now the stack is empty and curr points to T_r. Hence, the situation is the same as the situation for a tree with $k \leq n$. Therefore, T_r will be printed correctly.
So from the above analysis, we have-

   - T_l will be printed correctly i.e. satisfy (i), (ii) and (iii) (based on Induction Hypothesis)
   - r will be printed after T_l, before T_r and exactly once.
   - T_l will be printed correctly i.e. satisfy (i), (ii) and (iii) (based on Induction Hypothesis)

   Hence, it follows that traversal of tree T satisfies (i), (ii) and (iii).

Since the conditions hold in both the cases, the conditions are satisfied for traversal of any tree with $k = n + 1$.

Conclusion: Therefore, by principle of strong induction, the validity of (i), (ii) and (iii) and thereby, correctness of the function inorderTraversal is proved for all binary trees.

Q3(a) *Extend the idea in Q2, to write pseudo-code of non-recursive procedure for merge-sort using stacks. [Note that the code should be derived from the recursive version of merge sort. Unrelated non-recursive procedure for merge-sort will not get any marks].*

In the following pseudocode, we define a structure, namely subarray, consisting of integer field 'left', integer field 'right' and character field 'boolean.'
We assume a stack S of the instances of struct subarray. We do not repeat the implementation of the stack (as the basic idea remains same as covered in Lecture 8), but simply assume the stack operations.

Idea: Essentially, we try to imitate the working of system stack in the recursive mergesort algorithm, in the stack data structure.
In recursive mergesort algorithm, we would divide the array into half, call the mergesort function for each of the subarrays and then merge the two subarrays. Here, we store the divided subarrays in the stack. The struct variable, boolean is to keep track of which array has been divided into subarrays and which hasn't. So, we have the boolean variable as 'F', if the array has not yet been divided into subarrays and 'T' otherwise. We continuosly pop elements from the stack and do the following-

- If the subarray has 'F', then we assign it 'T', push it back to the stack, further create its sub-sub-arrays and push them as well in the stack.
- if the subarray has 'T', then we simply merge its two sub-sub-arrays.

```
1  struct subarray
2      int left
3      int right
4      char boolean
5
6  merge(A, i, k, j)
7      p=i, q=k+1, r=i, inv =0, count =0
8      while r≤j do
9          if p≤k and q≤j and A[p]>A[q]
10             B[count] = A[q]
11             q=q+1, count= count+1, r=r+1, continue
12         if p≤k and q≤j and A[p]≤A[q]
13             B[count] = A[p]
14             p=p+1, count= count+1, r=r+1, continue
15         if p≤k and q>j
16             B[count] = A[p]
17             p=p+1, count=count+1, r=r+1, continue
18         if p>k and q≤j
19             B[count] = A[q]
20             q=q+1, count= count+1, r=r+1, continue
21     //endwhile
22     count=0, r=i
23     while r≤j do
24         A[r]=B[count]
25         count= count+1, r=r+1
26     //endwhile
27
28 mergesort(a, left, right)
29     createStack(S)
30     subarray init;
31     init.left = left, init.right =right, init.boolean='F'
32     push(S,init)
33
34     while isEmptyStack(S)==False
35     do  subarray temp=pop(S)
36         left = temp.left, right = temp.right;
37
38         if left==right
39         then continue
40
41         if temp.boolean=='T'
42         then mid = left+⌊(right−left)/2⌋
43              merge(a,left,mid,right)
44              continue
```

```
45
46          if temp.boolean=='F'
47          then temp.boolean='T'
48               push(S,temp)
49               mid = left+⌊(right−left)/2⌋
50
51               subarray sub1
52               sub1.left = left
53               sub1.right = mid
54               sub1.boolean = 'F'
55               push(S,sub1)
56
57               subarray sub2
58               sub2.left = mid+1
59               sub2.right = right
60               sub2.boolean = 'T'
61               push(S,sub2)
62
63  main()
64      int a[n] //unsorted array of n integers given
65      mergesort(a,0,n-1)
66
67      count=0
68      while count<n
69      do  print(a[count])
```