ESO207A : Data Structures and Algorithms
Assignment 2 Solutions

Yatharth Goswami
Roll No. 191178

October 4, 2020

I Problem 1

This problem required us to implement Queue Data Structure using two stacks.

1.1 Part (a)

This part required us to describe our strategy for solving the task.

Given : Two Stacks S_1 and S_2. Implement a queue Q using these.
Solution : I will solve this question by making sure that the oldest entered element always remains at the top of Stack S_1 and during dequeue operation, I will always pop the element from stack S_1 and therefore this type of workflow will allow the oldest inserted element to be popped first, which is essentially what Queue does (FIFO).
Now, for implementing the above strategy, we will use stack S_2. Suppose, the above condition is true at some state of program and we want to perform the enqueue operation (say we want to enqueue an element x). So, for that we would perform the following in order:

- Pop each element from stack S_1 and push it into stack S_2 one after the other while S_1 becomes empty.
- Push the element x to stack S_2
- Pop each element from stack S_2 and push it into stack S_1 while S_2 becomes empty.

Notice after the end of the above sequence of moves the oldest element is still at the top and the newest entered element (x) is at the bottom of the stack S_1 (assuming the LIFO nature of operations in a stack). So, I will use the above strategy for implementing the operations related to Queue using two stacks. □

1.2 Part (b)

This part demanded from us to write pseudo-codes for various operations of Queue. I will implement all of them one by one by assuming that the given stacks support operations like push, pop, IsEmpty and IsFull.

IsEmptyQueue: For this operation, I will just check if stack S_1 is empty or not, since it contains the oldest element on the top.

Algorithm 1: Checking for emptiness of queue

```
struct Queue { stack $S_1$, $S_2$ }
Function IsEmptyQueue($Queue\ q$):
    if IsEmpty ($q.S_1$) then
        return true
    return false
```

IsFullQueue: For this operation, I will just check if stack S_1 is full or not, since it contains the all of the elements in the Queue at any state.

```
Algorithm 2: Checking for fullness of queue
struct Queue { stack $S_1$, $S_2$ }
Function IsFullQueue(Queue q):
    if IsFull (q.$S_1$) then
        return true
    return false
```

Dequeue : For this operation I will just pop from the stack S_1 and that will return me the oldest/first inserted element.

```
Algorithm 3: Perform Dequeue operation
struct Queue { stack $S_1$, $S_2$ }
Function Dequeue(Queue q):
    if IsEmptyQueue (q) then
        print ("Queue is Empty")
        return
    $x$ = pop(q.$S_1$)
    return $x$
```

Enqueue : For this operation, I will apply the strategy that I explained in the **Part (a)** of this problem.

```
Algorithm 4: Perform Enqueue operation
struct Queue { stack $S_1$, $S_2$ }
Function Enqueue(Queue q, element x):
    if IsFullQueue (q) then
        print ("Queue is already full")
        return
    while $q.S_1$ is not empty do
        $y$ = pop(q.$S_1$)
        push(q.$S_2$, $y$)
    push(q.$S_2$, $x$)
    while $q.S_2$ is not empty do
        $y$ = pop(q.$S_2$)
        push(q.$S_1$, $y$)
    return
```

Top : This function returns the value of the element present in the front of the queue without popping it. Therefore, for this operation I will just return the element present in the top of the stack S_1. I will assume that stack provided supports the operation *top*. If not, then we would have to first dequeue from the queue and then enqueue the same element.

Algorithm 5: Getting front element of the queue

```
struct Queue { stack $S_1$, $S_2$ }
Function Top(Queue q):
    if IsEmptyQueue (q) then
        print ("Queue is Empty")
        return
    $x$ = top(q.$S_1$)
    return $x$
```

Time Complexity Analysis : We will analyse the time complexity of each of the above defined functions one by one. I will assume here that stacks S_1 and S_2 have been implemented in standard manner and the time complexities of their operations are the same as discussed in class.

- **IsEmptyQueue :** Since checking for the emptiness of stack S_1 will take constant amount of time, therefore this complete function (Algorithm 1) will take constant time and hence the time complexity for checking emptiness of queue would be $O(1)$.
- **IsFullQueue :** Since checking for the fullness of stack S_1 will take constant amount of time, therefore this complete function (Algorithm 2) will take constant time and hence the time complexity for checking fullness of queue would be $O(1)$.
- **Dequeue :** Since popping an element from stack S_1 and checking for the queue's emptiness will take constant amount of time, therefore this complete function (Algorithm 3) will take constant amount of time and hence the time complexity for dequeue operation would be $O(1)$.
- **Enqueue :** For Algorithm 4, checking for fullness of queue takes constant amount of time, the *while* loops on line 6 and line 10 take amount of time proportional to the number of elements in the queue since each operation inside the loop takes constant amount of time and they themselves get repeated number of times equal to size of queue at that instant. Therefore, the overall time complexity of the Enqueue function becomes $O(n)$ where n is the size of queue.
- **Top :** Time Complexity of this function (Algortihm 5) relies on the fact if the provided stacks support operation such as top. If yes, then the time complexity would be $O(1)$, since getting the top element of stack takes constant amount of time. If no, then the running time would be the sum of running time of dequeue and enqueue operations, which implies the time complexity turns out to be same as enqueue operation which is $O(n)$.

1.3 Part (c)

This part of the problem demanded from us to argue about the correctness of the strategy deployed above. The main idea of the strategy used above is to use only one of the stacks to store the elements at any point of time and use the other stack as a buffer which is to be used when adding a new elemennt in the queue. Now, I will argue about the correctness of each function one by one.

- **IsEmptyQueue:** Since, the strategy described above uses only one of the stacks *say* S_1 to store the elements, checking for emptiness of that stack would suffice for checking emptiness of the queue as a whole. This is exactly what is done in Algorithm 1. Hence, proved. $\square$

- **IsFullQueue :** Since, the strategy described above uses only one of the stacks $say\ S_1$ to store the elements, checking for the fullness of that stack would suffice for checking fullness of the queue as a whole. This is exactly what is done in Algorithm 2. Hence, proved. □

For proving the correctness of implementation of queue we need to prove that dequeue always returns the oldest element entered in queue.

Observation : If at a particular state, stack S_1 holds the elements in chronological order with oldest being on top and newest being on bottom, performing any of the above five operations will not change this state/order (the new elements would also be in chronological order with oldest on top and newest on bottom).

Proof. The first two operations (*IsFullQueue, IsEmptyQueue*) do not change the original queue at all therefore they will not change the chronological state of queue. Now, we will check for the other three operations.
Dequeue : This operation will take away the oldest/top element on the stack S_1 and since we are not disturbing any other element the other elements will remain in same order and hence the complete queue will still be in chronological order with oldest on top and newest on bottom.
Enqueue : Notice the fact that moving all elements from one stack to another reverses the chronological order of elements. For proving this without loss of generality, let's assume that initial chronological order was such that oldest was on top and newest was on bottom, now the top elements would be popped first and pushed into the empty stack so that they occupy the bottom of that stack and newer elements would be popped later and hence would occupy the top of the new stack which leads to reversal of the order. Now, applying Algorithm 4 here would first lead to reversal of order of elements in stack S_2 with newer elements on top and older at bottom. Now adding a new element (x) to the top would not change the order of remaining elements and hence the order of the complete queue would be maintained. Now, another transfer from stack S_2 to S_1 would again change the order and hence the oldest element would again be on top and newerst on bottom. Hence, the original chronological order before applying enqueue operation is still maintained.
Top : This operation will return the oldest/top element on the stack S_1 and since we are not disturbing any element, all elements will remain in same order and hence the complete queue will still be in chronological order with oldest on top and newest on bottom. □

Hence, using the above observation we can say that since at the start of the queue's timeline since there was no element and it is chronologically ordered, the queue will remain ordered after any operation. Hence, the dequeue operation would always fetch the first entered element which is the characteristic property of queues and hence the algorithm is correct.

II Problem 2

2.1 Part (a)

This problem required us to write the pseudo-code for in-order traversal of binary tree in a recursive manner. I will use the linked representation of binary tree to implement the program as used in the class. The main idea of the algorithm is to move to the subtree rooted at left child of the current node first and then print the current node's value and then moving to the subtree rooted at right child of the current node.

Algorithm 6: In-order traversal of a binary tree in recursive manner

Input: Binary Tree T in linked format with T representing root of the tree
Output: Prints the binary tree in in-order traversal order

```
Function InorderTraversal(Binary_Tree T):
    if T is not nil then
        InorderTraversal ($T.lchild$)
        print ($T.val$)
        InorderTranversal ($T.rchild$)
    return
```

2.2 Part (b)

This part required us to write the pseudo code written above in non-recursive format with the help of stack. I will use the same linked representation of binary tree as used in the previous part for solving this. Also I will assume that the stack operates in the same way as discussed in lectures and provides the basic operations. The main idea of the algorithm is to visit the leftmost part of the tree and push the nodes visited while traversing to the leftmost part into the stack and then printing the value of the leftmost non visited node and then moving to the right subtree in a similar way.

Algorithm 7: In-order traversal of a binary tree in non-recursive manner

Input: Binary Tree T in linked format pointing to root of the tree and a Stack S
Output: Prints the binary tree in in-order traversal order in non-recursive manner

```
Function InorderTraversalStack(Stack S):
    $temp = pop(S)$
    while S is not empty OR temp is not nil do
        while temp is not nil do
            push ($S$, temp)
            $temp = temp.lchild$
        $temp$ = pop ($S$)
        print ($temp$)
        $temp = temp.rchild$
    return
Function CreateStack(Stack S, Binary_Tree T):
    push ($S$, $T$)
    InorderTraversalStack ($S$)
    return
```

2.3 Part (c)

This part required us to prove the correctness of the algorithm given above. We will use the following specification that traversing a tree in In-order traversal will always print the leftmost node which is not printed yet. So, we will try to prove that if we traverse tree according to the above algorithm we will always print the nodes of tree according to the above specification and hence it's correctness.

Observation 1 : After the loop at line 4 stack S contains the leftmost node of the binary tree rooted at $temp$.

Proof. It is easy to proof this observation by noticing that at each step we are moving to the left child of current node and the loop stops when there is no left child of that node and hence that node has to be the leftmost node in the whole tree. Since, we are also filling the stack simultaneously so just before the loop terminates it stores this node (which has no left child) and hence the top of the stack contains the leftmost node in this tree. □

We will prove the correctness of loop at line 3 using loop invariant. The specific loop invariant in this case would be that the value which was printed inside previous iteration was the leftmost among the nodes that were not printed yet. Let's prove this invariant now.

Initialisation : At the start of the loop, there was no previous loop so the invariant is trivially true.

Maintenance : Suppose the following invariant holds before line 4. Notice that in the end of execution of previous loop, the new value of $temp$ is the right child of the last printed node, and since the last printed node was the leftmost among the non-printed nodes, therefore by the property of binary tree the tree rooted at right child of this node would be the one which contains the leftmost non-printed node in the tree. Now, using **Observation 1** the stack contains the leftmost node in this subtree at the top after loop at line 4 and print this node's value next. Hence, the invariant holds after the execution of the loop.

Before proceeding with the termination part of the invariant, I would prove that at the end of the program every node of the tree gets printed exactly once.

Proof. We will prove this using two observations.

- It's trivial to see that if a particular node gets into the stack, it's both child nodes also get into the stack. And if a node doesn't get printed, it means it never entered the stack. This is only possible if either it doesn't have a parent (root) or the parent doesn't enter into the stack. The first case is not possible since the root always enters into the stack first. The second implies that parent doesn't enter the stack and it's only possible if it's parent doesn't enter the stack and in this way we can ascend the tree to imply that root didn't enter the stack which is contradiction. Hence, every element enters into the stack and gets printed atleast once.
- According to the above loop invariant every loop prints the lowest amongst the non-printed nodes. Therefore a node can't get printed twice as after getting printed once it would not be in set of non-printed nodes.

Hence, proved that all the nodes get printed exactly once after at the end of the program. □

Termination : At this stage the value of temp would be nil and stack S would be empty. And since every node in the tree got printed by the above observations, they would get printed in the right order as specified in the specification also by the invariant property. Hence proved, that above algorithm satisfies the specification of inorder traversal.

III Problem 3

3.1 Part (a)

```
Algorithm 8: Non-recursive Merge Sort using Stacks $A$
    Input: Array $A$ containing n integers
    Output: Number of inversions present in $A$
Function MergeSort(Array A, Length of Array $n$):
    $S$ = Empty Stack that can store integers
    $S$.push (0), $S$.push (0), $S$.push $(n-1)$
    while $S$ is not empty do
        $(x,\ y,\ z)\ =\ (S.pop(),\ S.pop(),\ S.pop())$
        if $z == 1$ then
            Merge(A, y, x)
        else if $y < x$ then
            $mid = \lfloor \frac{x+y}{2} \rfloor$
            $S.push(1)$, $S.push(y)$, $S.push(x)$
            $S.push(0)$, $S.push(y)$, $S.push(mid)$
            $S.push(0)$, $S.push(mid+1)$, $S.push(x)$
    return
Function Merge(Array A, start $l$, end $r$):
    $sorted$ = Empty array of size $r-l+1$, $i = l$, $j = mid+1$, $k = 0$
    while $i \leq mid$ AND $j \leq r$ do
        if A[i] <= A[j] then
            $sorted[k] = A[i]$
            $k = k+1$
            $i = i+1$
        else
            $sorted[k] = A[j]$
            $k = k+1$
            $j = j+1$
    while $i \leq mid$ do
        $sorted[k] = A[i]$
        $k = k+1$
        $i = i+1$
    while $j \leq r$ do
        $sorted[k] = A[j]$
        $k = k+1$
        $j = j+1$
    for $i = l$ to $r$ do
        $A[i] = sorted[i-l]$
    return

```

This problem demanded us to write the pseudo code for implementing non-recursive merge sort using a single stack. I will assume that the stacks available support basic operations like Push, Pop and IsEmpty.

The main idea of the algorithm involves using the bottom-up method in the sense that we move from bottom of the tree in divide and conquer to the top non-recursively by storing the array ranges in the stack. Therefore, stack will contain ranges with smallest ranges on the top of stack and biggest ranges on the bottom. We would be merging these ranges from top of the stack to bottom and in the end we would get the sorted array.

I will represent every array range in form of a tuple of integers (a, l, r) such that l and r are the start and end of the range and a is a boolean value which is true if a range is ready to be merged (both the half-arrays have been sorted) and false otherwise. Since, here we were only allowed to use stacks which support integers therefore instead of storing the ranges as tuples, I will push these three values one by one into stack and pop them in the same order and boolean a would be true iff it is 1 and false if it is 0.

3.2 Part (b)

Implementation based.

3.3 Part (c)

Implementation based.