ESO207A
Assignment 2

Rahul Sethi
190668

Q1

(a)

Implementation of a Queue using two Stacks:
We begin with two empty stacks, S_1 and S_2. To enqueue an element x, we push it to S_1. The dequeue operation is slightly more involved:

1. If S_2 is not empty, pop an element from S_2 and return it.
2. If S_2 is empty, pop all elements of S_1 and push them into S_2. Lastly, pop an element from S_2 and return it.

(b)

Complexity of Queue Operations:

- $\textsc{Enqueue}(S_1, S_2, x)$ takes $\mathcal{O}(1)$ time, since it only involves pushing the element to S_1. This is a *constant-cost* operation.
- $\textsc{Dequeue}(S_1, S_2)$ takes $\mathcal{O}(n)$ time, where n is the length of the queue. In the worst-case, we pop all n elements from S_1 before pushing them to S_2.
- $\textsc{IsEmpty}(S_1, S_2)$ and $\textsc{IsFull}(S_1, S_2)$ take $\mathcal{O}(1)$ time.

Algorithm 1 $\textsc{MakeQueue}$

```
stack S1 ⟵ [ ]        ▷ initialize two empty stacks
stack S2 ⟵ [ ]
```

Algorithm 2 $\textsc{Enqueue}(S_1, S_2, x)$

Input: Stacks S_1, S_2 and element x

```
if not IsFull(S1, S2) then
    S1.push(x)
else
    return
```

Algorithm 3 DEQUEUE(S_1, S_2)

Input: Stacks S_1, S_2

Output: The dequeued element

```
1:  if IsEmpty(S1, S2) then
2:      return                                  ▷ empty queue
3:  if not S2.empty() then
4:      z ⟵ S2.pop()
5:      return z
6:  else                                        ▷ S2 is empty
7:      while not S1.empty() do
8:          z ⟵ S1.pop()
9:          S2.push(z)
10:     z ⟵ S2.pop()
11:     return z
```

Algorithm 4 ISEMPTY(S_1, S_2)

Input: Stacks S_1, S_2

```
if S1.empty() and S2.empty() then
    return true
else
    return false
```

Algorithm 5 ISFULL(S_1, S_2)

Input: Stacks S_1, S_2

```
if S1.full() then
    return true
```

(c)

We design a FIFO(*First-In-First-Out*) data structure, i.e. a queue, using two LIFO(*Last-In-First-Out*) data structures (stacks). Showing the correctness of this implementation would involve proving the FIFO nature of operations, and we start with the following observations:

1. Invariant ϕ_1: Either S_2 is empty, or S_2.top() is the first element of the queue.
2. Invariant ϕ_2: If S_1 is not empty, S_1.top() is the last element of the queue.
3. Invariant ϕ_3: If S_2 is empty, S_1 contains the entire queue descending sorted according to its indices from top to bottom of the stack S_1.

Correctness of DEQUEUE

Claim: DEQUEUE(S_1, S_2) dequeues and returns the first element of the queue.
Proof: If S_2 is not empty, S_2.top() is popped and returned. The correctness of this operation is guaranteed by ϕ_1 defined above. On the other hand, if S_2 is empty, the while loop in Lines 5-8 pops elements from S_1 and pushes them into S_2. This is accompanied by an obvious reversal in ordering. After the loop is executed, S_2.top() is the first element of the queue (due to ϕ_2 and ϕ_1).[1]

[1] ϕ_1 is in fact a consequence of ϕ_2, but they've been listed separately for clarity.

Correctness of Enqueue

Claim: Enqueue(S_1, S_2, x) enqueues x into the queue, i.e. x is the last element in the queue.
Proof: Enqueue(S_1, S_2, x) involves just one operation, i.e. S_1.push(x) which adds x on the top of stack S_1. Using invariant ϕ_2, it is clear that this is the last element of the queue.

■

Q2

(a)

Algorithm 6 InOrder(T)

Input: T, the root node of the binary tree

if $T \neq$ nil **then**
    InOrder(T.left)
    print(T.val)
    InOrder(T.right)
return

For a given node x in the tree, the following attributes are assumed:

- x.left - the left child of x
- x.right - the right child of x
- x.val - the (integer) value contained in x

(b)

Algorithm 7 InOrder-Iterative(T)

Input: T, the root node of the binary tree

stack $S \longleftarrow [\,]$ ▷ initialize empty stack S
$current \longleftarrow T$
while S is not empty **or** $current \neq$ nil **do**
    if $current \neq$ nil **then**
        S.push($current$)
        $current \longleftarrow current$.left
    else
        $current = S$.pop()
        print($current$.val)
        $current \longleftarrow current$.right

(c)

Definition 1

The depth of a node in a binary tree is the number of edges from the root to the node. In addition, the **depth of a tree** refers to the depth of the deepest leaf. Hereafter, the depth of the tree rooted at node T is represented by d_T.

Definition 2

In-order (depth-first) traversal of a binary tree T, refers to recursively traversing the left sub-tree, followed by the root node T, and finally the right sub-tree.

Definition 3

The concatenation operator $\oplus$ concatenates two lists while maintaining their individual order. For example, $[z_1, z_2, z_3] = [z_1, z_2] \oplus [z_3]$.

Proof of Correctness of InOrder-Iterative(T)

We shall prove the correctness of the code in part (b) using *strong induction on the depth of the tree.*

Base Cases: ($d_T = 0$ or 1)
Let's quickly go over the uninteresting case of $d_T = 0$, i.e. the binary tree containing only the root node T. *current* is initialized to T (Line 2), and pushed into the stack S in Line 5. Since T has no left (or right) child, *current* becomes nil and we print S.pop() $= T$ in Line 9. Now, S is empty and *current* $=$ nil, so the program terminates with the desired output.

If $d_T = 1$, the tree (with root node T) can be represented by *Figure 1*, where A and B are T's left and right children respectively.

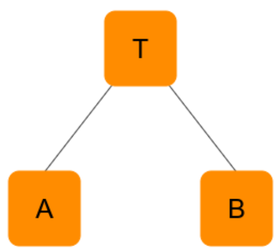

Figure 1: Drawn using yEd Live

It is easy to see that the first time *current* $=$ nil is when the stack $S = [T, A]$.[2] Here, A and T are popped & printed in order, and *current* $= B$. B is pushed onto the stack, and popped & printed right after. All in all, $[A, T, B]$ is printed. This completes the in-order traversal.

Induction Hypothesis:
InOrder-Iterative(T) correctly traverses (in-order) binary trees of depth $= 0, 1, ...n-1$, rooted at T.

Observation 1:
$\forall$ T, when InOrder-Iterative(T) terminates, the corresponding stack $S = \phi$ and *current* $=$ nil. If $S \neq \phi$ or *current* $\neq$ nil, we couldn't possibly have exited the while loop in Lines 3-12 - thus, the call InOrder-Iterative(T) would not have terminated.

Continued on next page

[2]See the appendix for details about the notation used for representing stacks in this document. Familiarity with the same will be assumed hereafter.

Proof:
Consider the following tree of depth n with root node T, and left and right sub-trees denoted by A and B respectively. It is easy to see that the sub-trees A and B have depth at most $n-1$. Also, at least one of A and B has depth exactly $n-1$, else the tree rooted at T would not have had depth n.

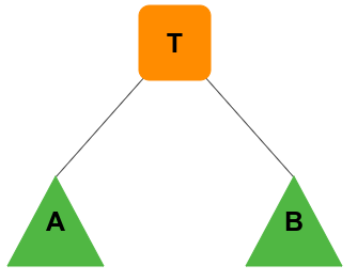

Figure 2: Drawn using yEd Live

In Line 2, T is pushed onto stack S and $S = [T]$. Consider a new stack $S' = S \backslash \{T\}$, i.e. $S' = \phi$ initially. In the next iteration of the while loop, the left sub-tree's root node(A) is pushed onto the stack. So $S = [T, A]$ and $S' = [A]$. Clearly, this is the same as calling InOrder-Iterative(A), i.e. in-order traversal on the left sub-tree only, using stack S'. Let the output of InOrder-Iterative(A) be the list L_1. Using the induction hypothesis on A (tree of depth at most $n-1$), we are guaranteed that L_1 is the correct in-order traversal of A. Moreover, when this traversal is complete, $S' = \phi$, $current = \text{nil}$ (from Observation 1) and $S = [T]$ (by definition). We shall forget about S' beyond this stage.

As $S \neq \phi$ and $current = \text{nil}$,[3] the root node T is popped & printed and $current \longleftarrow T.\text{right} = B$. For consistency in notation, we denote this output by $L_2 = [T]$. Now, B is pushed onto the empty stack S (Line 5), and the rest of the program is identical to InOrder-Iterative(B). The output of this traversal is denoted by L_3.

Clearly, the output of InOrder-Iterative(T) is given by $L_1 \oplus L_2 \oplus L_3$, where,

- L_1 is the output of InOrder-Iterative(A)
- $L_2 = [T]$
- L_3 is the output of InOrder-Iterative(C)

The output $L_1 \oplus L_2 \oplus L_3$ exactly matches the recursive definition of in-order traversal, hence proving the correctness of InOrder-Iterative(T). Lastly, like any other traversal, the time complexity of this algorithm is $\mathcal{O}(n)$ (we visit every node at most twice).

■

Q3

(a)

The time complexity is $\mathcal{O}(n \log n)$, same as the recursive version.

Continued on next page

[3] Slight abuse of notation. *current* is same in the context of InOrder-Iterative(a) and InOrder-Iterative(T) both. This is because we view InOrder-Iterative(a) as a part of InOrder-Iterative(T).

Algorithm 8 ITERATIVE-MERGESORT()

Input: Array A of size n

```
function MERGESORT(A, n)
    stack S ⟵ [ ]
    m ⟵ (1, n, False)                                ▷ tuple of the form (start, end, flag)
    S.push(m)
    while not S.empty() do
        m ⟵ S.pop()
        if m[3] is True then                         ▷ flag is True → sorted halves, ready for MERGE
            MERGE(A, m[1], m[2])
        else
            if m[1] < m[2] then
                mid ⟵ ⌊(m[1]+m[2])/2⌋
                S.push((m[1], m[2], True))
                S.push((m[1], mid, False))
                S.push((m[1] + 1, m[2], False))
    return
function MERGE(A, start, end)
    B[i] ⟵ 0 for all 1 ≤ i ≤ n
    mid ⟵ ⌊(start+end)/2⌋
    p ⟵ start
    q ⟵ mid + 1
    r ⟵ start
    while p ≤ mid and q ≤ end do
        if A[p] ≤ A[q] then
            B[r] ⟵ A[p]
            r += 1
            p += 1
        else
            B[r] ⟵ A[q]
            r += 1
            q += 1
    while p ≤ mid do                                 ▷ right subarray exhausted
        B[r] ⟵ A[p]
        r += 1
        p += 1
    while q ≤ end do                                 ▷ left subarray exhausted
        B[r] ⟵ A[q]
        r += 1
        q += 1
    while start ≤ i ≤ end do                         ▷ copy sorted sequence to original array
        A[i] ⟵ B[i]
        i += 1
    return
MERGESORT(A, n)                                      ▷ implements above procedure
```

Appendix

Stack Operations

If S is a stack,

- S.top() is the topmost element in the stack
- S.pop() pops S.top() and returns it
- S.push(x) pushes the element x onto the stack
- S.empty() returns true if stack S is empty
- S.full() returns true if stack S is full, i.e. it is not possible to push more elements onto S

Notation

1. We represent a stack as a list $[a_1, a_2, ...a_n]$ where a_1 and a_n are the stack's bottom and top elements respectively.
2. A tree and its root node are represented by the same symbol, i.e. T may refer to the tree rooted at node T, or the node T itself (obvious from the context).
3. The pseudo-code in Algorithm 8 uses three-element tuples of the form $m = (m[1], m[2], m[3])$, where $m[i]$ denotes the i^{th} element of tuple m. Moreover, all $m[i]$ need not be of the same data-type. We use tuples of type (int, int, bool).