Tutorial 3 Exercise Solutions

COMP526: Efficient Algorithms

21–22 October, 2024

Exercise 1. Recall that a Stack is an ADT that supports the functions Push, Pop, Empty, and Top. A Queue supports the methods Enqueue and Dequeue (among others). Suppose you are given two Stack instances, A and B. How could you use A and B to simulate the behavior of a Queue? That is, how can you implement Enqueue and Dequeue using *only* A and B, and the associated Stack methods for A and B?

Solution. In the specification of both the Stack and Queue ADTs, the state of the ADT is represented by a sequence S of elements stored in the ADT. In the case of a Stack, both Push and Pop operations modify the (right) end of S by either appending a new element (Push) or removing the last element (Pop). On the other hand, in the case of a Queue, Enqueue *pre*pends an element to S, while Dequeue removes the last element from S. Since S could represent the state of either a Stack or a Queue, we just need to figure out how to simulate Enqueue and Dequeue on S. For concreteness, we will use Stack A to store the contents of the Queue between method calls, and use B as an auxiliary Stack to help us perform the Queue operations.

The case of Dequeue is straightforward because Pop and Dequeue are formally the same: in both cases $Sx \mapsto S$ and the value x is returned. Thus, we can easily implement Dequeue as follows.

```
procedure Dequeue
    return A.Pop()
end procedure
```

The Enqueue(x) procedure requires a little more thought because we must access the *bottom* of the Stack to prepend an element to S. The idea is to transfer the elements from A to B, then Push(x) to A, and transfer the elements from B back to A so that x is on the bottom of A. This will have the same effect as Enqueue(x), as $S \mapsto xS$.

```
procedure Enqueue(x)
    while not A.Empty() do                  ▷ transfer elements from A to B
        B.Push(A.Pop())
    end while
    A.Push(x)
    while not B.Empty() do                  ▷ transfer elements from B back to A
        A.Push(B.Pop())
    end while
end procedure
```

You should verify for yourself that the contents of A remain in the correct order after transferring the elements to B then back to A. □

Exercise 2. STACKs and QUEUEs are limited in that in both cases, elements are only added to one "side" of the sequence of elements, and elements are only removed from one side. In the case of STACKs, all modifications affect only the top of the STACK. For QUEUEs, elements are enqueued to the "back" and dequeued from the "front." We can generalize both ADTs to the DEQUE (pronounced "deck") ADT that allows modifications (additions and removals) to both "ends" of the sequence of elements stored in the ADT. Formally, we can represent a DEQUE as follows:

- The state of the DEQUE is a sequence S, initially $S = \varnothing$
- APPEND(x) modifies $S \mapsto Sx$
- APPENDLEFT(x) modifies $S \mapsto xS$
- POP() modifies $Sx \mapsto S$ and returns x
- POPLEFT() modifies $xS \mapsto S$ and returns x

How could you implement a DEQUE with an array such that all operations can be performed in $O(1)$ time? How you determine if the DEQUE is full? (You may assume that the size of the array is fixed so that we don't need to worry about resizing.)

Solution. We can use the same ideas as our QUEUE implementation where we use a *circular* array. That is, if the array has capacity n, we perform index arithmetic $\bmod n$, so that index 0 is after index $n-1$. As with the QUEUE implementation, we keep track of a *head* index and at *tail* index that refer to the elements at the "right" and "left" ends of the DEQUE elements, respectively. Here is an implementation (that ignores resizing, and issues with empty/full DEQUEs).

```
class ARRAYQUEUE
    a ← new array, size n
    head, tail, size ← 0
    procedure APPEND(x)
        size ← size + 1
        a[head] ← x
        head ← head + 1 mod n
    end procedure
    procedure APPENDLEFT(x)
        size ← size + 1
        tail ← tail − 1 mod n
        a[tail] ← x
    end procedure
    procedure POP
        size ← size − 1
        head ← head − 1 mod n
        return a[head]
    end procedure
    procedure POPLEFT
        size ← size − 1
        tail ← tail + 1 mod n
        return a[tail − 1 mod n]
    end procedure
end class
```

Note that in the pseudocode above, the case *head* = *tail* occurs both when the DEQUE is empty and when it is full (i.e., it stores n elements, where n is the size of the array). Thus, in order to check if the DEQUE is full, we should check if $size = n$. □

Exercise 3. In Lecture 05, we described the "bubble up" procedure for adding a new element to a heap:

```
1: procedure INSERT(p)
2:     v ← new vertex storing p
3:     u ← first vertex with < 2 children
4:     add v as u's child
5:     PARENT(v) ← u
6:     while v is not the root and value(v) < value(u) do
7:         SWAP(value(v), value(u))
8:         v ← u
9:         u ← PARENT(v)
10:    end while
11: end procedure
```

Prove that the INSERT procedure is correct: That is, argue that if T was a heap before calling INSERT(p), then T is a heap after calling T.

Solution. Recall that a heap T must satisfy two properties:

(A) T is a complete binary tree

(B) For every vertex u storing the value p_u and child v storing the value p_v, we have $p_u \leq p_v$.

Property (A) is guaranteed by the choice of where the vertex v is added to the tree. Thus, we focus on establishing (B). To this end, we argue that the following loop invariant holds:

loop invariant The only violation to Property (A) (if any) occurs at vertex v with u = PARENT(v).

We argue this loop invariant by induction. The base case holds because T satisfied the heap property before the insertion, and v (the new vertex) doesn't have any children. Thus, the only possible violation is between v and u = PARENT(v).

For the inductive step, suppose loop invariant holds after iteration i of the loop. If v doesn't violate (B) with PARENT(v), then the procedure terminates and we are done. Suppose that v does violate (B) with its parent. Then in iteration $i+1$, the values of v and u = PARENT(v) are swapped in line 7. After the swap, $p_u < p_v$, so there is no longer a violation of (B) between v and u. Further, since p_u is smaller than its previous value, u cannot violate (B) with its other child. Since no other values in the heap change, the only possible violation of (B) is between u and PARENT(u), which are updated to v and u (respectively) in lines 8–9. Thus, the invariant holds also after iteration $i+1$, as desired.

Finally, we note that the process terminates when either v is the root, or v no longer violates (B) with its parent. Thus, the procedure results in a heap. □