Homework 3 sample solution

Due 09/25/2024

September 26, 2024

1. Give pseudocode for an algorithm that accepts a stack *stk* of size n and an integer k between 0 and n that modifies *stk* so that its top k elements are reversed, using only a queue for storage. (You may not declare any variables other than a single queue.)

   Input: *stk*: stack with n elements
   Input: n: size of *stk*
   Input: k: integer between 0 and n
   Output: modified *stk* so that top k elements are reversed
   Algorithm: StackTopReverse
   $q =$ Queue()
   for $i = 1$ to k **do**
   | q.Enqueue(*stk*.Pop())
   end
   for $i = 1$ to k **do**
   | *stk*.Push(q.Dequeue())
   end
   return *stk*

   It's also possible to avoid declaring i if you decrement k to 0 then dequeue until the queue is empty, but this is not necessary.

2. Give pseudocode for an algorithm that accepts a stack *stk* of size n and integers i and j such that $0 \leq i \leq j \leq n$ and modifies *stk* so that all of its entries between indexes i and j (including j but excluding i) are reversed, using only a queue for storage.

   Note that if $i = 0$, the resulting stack will have the top j entries in reverse order. Also, if $j = i$ or $j = i + 1$, the stack should retain all elements in the same order.

   Hint: you may call your solution to problem 1 as a subroutine.

   Expected answer:

Input: *stk*: stack with n values
Input: n: size of *stk*
Input: i, j: integers such that $0 \leq i \leq j \leq n$
Output: modification of *stk* where values in the range $(i, j]$ are reversed

1 **Algorithm:** StackRangeReverse
2 StackTopReverse(*stk*, j)
3 StackTopReverse(*stk*, $j - i$)
4 StackTopReverse(*stk*, j)
5 **return** *stk*

The question was posed ambiguously—i and j are meant to be indexes relative to the top of the stack (same as questions 1 and 3), but this is not clear from the question. If you interpret i and j as array indexes (i.e., counting from the bottom of the stack), then j in lines 2 and 4 should be exchanged with $n - i$ (possibly $n - 1 - i$ depending on whether the first index is 0 or 1).

Inlining the solution to Q1 (provided the queue is reused) is also reasonable, and there may be other valid approaches.

3. Give pseudocode for an algorithm that accepts a stack *stk* of size n and integers i and j such that $1 \leq i \leq j \leq n$ and modifies *stk* so that indexes i and j (counting from the top) are swapped, using only a queue for storage.

   Hint: you may call your solution to problem 2 as a subroutine.

Input: *stk*: stack with n values
Input: n: size of *stk*
Input: i, j: integers such that $1 \leq i \leq j \leq n$
Output: modification of *stk* where indexes i and j are swapped

Algorithm: StackSwap
StackRangeReverse(*stk*, $i - 1$, j)
if $i \neq j$ **then**
    StackRangeReverse(*stk*, i, $j - 1$)
end
return *stk*

The two calls to problem #2 can be in either order. Checking for $i = j$ is nice but not required for full credit. Other approaches are possible.

4. Give pseudocode for a post-order tree traversal for an m-ary tree stored in firstChild–nextSibling form.

   Solution #1:

```
Input: node: node in an m-ary tree in
       firstChild–nextSibling form
Algorithm: PostOrder
if node = nil then
    return
end
child = node.firstChild
while child ≠ nil do
    PostOrder(child)
    child = child.nextSibling
end
Process/print node
```

Solution #2:

```
Input: node: node in an m-ary tree in
       firstChild–nextSibling form
Algorithm: PostOrder
if node = nil then
    return
end
PostOrder(node.firstChild)
Process/print node
PostOrder(node.nextSibling)
```

The code could also test for null pointers before recursing instead of using null as the base case.

The final recursive call to nextSibling in solution #2 represents returning to your parent and having the parent recursively explore the next sibling. This can only happen after the current *node* has been processed or printed, as a post-order traversal does not return until after processing the node.

5. Answer the following questions about a modification to a BST so that it can return the min value in $\Theta(1)$ time.

   (a) What additional data members would you store in the BST? Your answer should indicate what these data members represent and how they can be used to identify the min in $\Theta(1)$.

   Such a BST must store the min value (or a pointer to the node containing it). To return the min, return this value.

   (b) How would you modify insertion so that your data members are updated appropriately? Pseudocode for a basic BST insertion has been given below. Your modification should exhibit the same time complexity.

   Your answer may be an English-language description of how you would modify this pseudocode, or it may be the modified pseudocode.

```
Input: ins: new data value to insert
Input: node: current BST node (originally root)
Input: BST.Insert
if ins ≤ node.value then
    if node.left = nil then
        node.left = Node(ins)
        node.left.parent = node
    else
        BST.Insert(ins, node.left)
    end
else
    if node.right = nil then
        node.right = Node(ins)
        node.right.parent = node
    else
        BST.Insert(ins, node.right)
    end
end
```

Before inserting as a left child (or honestly, anywhere), check whether *ins* is less than the current min. If so, update the min to *ins*.

(c) How would you modify deletion? Your modification should exhibit the same time complexity as ordinary deletion.

```
Input: victim: BST node to be deleted
Input: BST.Delete
Let children be the number of non-nil children of victim
if children = 0 then
    if victim.parent ≠ nil then
        Set victim.parent's matching child pointer to nil
    end
    delete victim
else if children = 1 then
    Let child be the child of victim
    if victim.parent ≠ nil then
        Set victim.parent's matching child pointer to child
    end
    child.parent = victim.parent
    delete victim
else
    lhsMax = victim.left
    while lhsMax.right ≠ nil do
        lhsMax = lhsMax.right
    end
    Swap victim.data and lhsMax.data
    BST.Delete(lhsMax)
end
```

At the end (or maybe after modifying the pointers but before deallocating the node), check whether the *victim* contains the min value. If so, scan for the new min: start at the root and follow $left$ pointers until you reach a null pointer. This final step will take $O(h)$ time, which does not increase the asymptotic complexity of Delete (only its coefficient).