Solutions to Exercise Set 3

March 28, 2009

Chapter 5

R-5.1)

The size of the stack is $25 - 10 + 3 = 18$.

R-5.4)

If the stack is empty, then return (the stack is empty). Otherwise, pop the top element from the stack and recur.

R-5.7)

The size of the queue is $32 - 15 + 5 = 22$.

R-5.10)

D.addLast(D.removeFirst())
D.addLast(D.removeFirst())
D.addLast(D.removeFirst())
Q.enqueue(D.removeFirst())
Q.enqueue(D.removeFirst())
D.addFirst(Q.dequeue())
D.addFirst(Q.dequeue())
D.addFirst(D.removeLast())
D.addFirst(D.removeLast())
D.addFirst(D.removeLast())

C-5-1)

The solution is to actually use the queue Q to process the elements in two phases. In the first phase, we iteratively pop each the element from S and enqueue it in Q, and then we iteratively dequeue each element from Q and push it into S. This reverses the elements in S. Then we repeat this same process, but this time we also look for the element x. By passing the elements through Q and back to S a second time, we reverse the reversal, thereby putting the elements back into S in their original order.

C-5.3)

$x \leftarrow S.pop()$
if $x < S.\top()$ **then**
  $x \leftarrow S.pop()$
end if

Note that if the largest integer is the first or second element of S, then x will store it. Thus, x stores the largest element with probability 2/3.

C-5.6)

Let S be an array-based stack with capacity n. We will store integer triples (i, j, k) in the stack S, where each such triple stands for the fact that "$A[i, j] = k$," for a cell $A[i, j]$ that we wish to actually use. But instead of storing k in this cell, we instead store the index, m, of where the triple (i, j, k) is stored in the stack S. To read a cell, $A[i, j]$, then, we set $m = A[i, j]$ (which might be garbage), and, if $0 \leq m \leq top$, then we go to $S[m]$ and read the triple (i', j', k) that is there. If $i' = i$ and $j' = j$, then we know that this $A[i, j]$ was previously assigned a value and that value is k. Otherwise, the real value of $A[i, j]$ is 0. To write the value k to a cell $A[i, j]$ we push (i, j, k) on S and set $A[i, j] = top$.

Chapter 7

R-7.7)

The running time is $O(n_v)$, where n_v is the number of nodes in the subtree rooted at node v.

R-7.11)

It is not possible for the postorder and preorder traversal of a tree with more than one node to visit the nodes in the same order. A preorder traversal will always visit the root node first, while a postorder traversal node will always visit an external node first.

It is possible for a preorder and a postorder traversal to visit the nodes in the reverse order. Consider the case of a tree with only two nodes.

R-7.12)

It is not possible for the post and pre order traversals to visit the nodes of a proper binary tree in the same order for the same reason in the previous question.

It is not possible for the post and pre order traversals to visit the nodes of a proper binary tree in the reverse order. Let a be the root of a proper binary tree and let T_1 and T_2 be the left and right subtrees. A postorder traversal would visit the postorder traversal of T_1, the postorder traversal of T_2 and then node a while the preorder traversal would visit node a, the preorder traversal of T_1 and then the preorder traversal of T_2. Clearly the postorder and preorder traversals cannot be the reverse of each other since in both cases, all the nodes of T_1 are visited before all the nodes of T_2.

R-7.13)

The running is $O(n)$ since parentheticRepresentation is just a preorder traversal with output.

R-7.17)

grades hw1 hw2 hw3 homeworks/ pr1 pr2 pr3 projects/ cs016/ buylow sellhigh papers/ market demos/ projects/ grades cs252/ /user/rt/courses

R-7.20)

1. 3 1 + 3 x 9 5 - 2 + - 3 7 4 - x 6 + -
2. $((((3+1)x3)/((9-5)+2)) - ((3x(7-4))+6))$

C-7.1)

$post(v) = desc(v) - depth(v) + pre(v)$. To prove this, consider the difference, $post(v) - pre(v)$. This difference is equal to all the nodes counted by postorder but not preorder (the descendents of v, whose number is $desc(v)$) minus all the nodes counted by preorder but not postorder (the ancestors of v, whose number is $depth(v)$).

C-7.2)

We can accomplish the task of printing the string stored at v along with the height of the subtree rooted at v by using a postorder traversal. During this traversal, we will find the height of each node. The height for a node v will be 0 if v is external and one more than the height of the max child for an internal node. Then, we can simply print out the string at v and its height if v is internal.

C-7.3)

```
preorderNext(Node v)
  if v.isInternal() then
    return v's left child
  else
    Node p = parent of v
    if v is left child of p then
      return right child of p
    else
```

```
        while v is not left child of p do
          v = p
          p = p.parent
        end while
        return right child of p
      end if
    end if
```

```
inorderNext(Node v)
    if visInternal() then
      return v's right child
    else
      Node p = parent of v
      if v is left child of p then
        return p
      else
        while v is not left child of p do
          v = p
          p = p.parent
        end while
        return p
      end if
    end if
```

```
postorderNext(Node v)
    if visInternal() then
      p = parent of v
      if v = right child of p then
        return p
      else
        v = right child of p
        while v is not external do
          v = leftchildofv
        end while
        return v
      end if
    else
      p = parent of v
      if v is left child of p then
        return right child of p
      else
        return p
      end if
    end if
```

The worst case running times for these algorithms are all $O(n)$ where n is the

height of the tree T.

C-7.4)

This can be done using a preorder traversal. When doing a "visit" in the traversal, simply store the depth of the node's parent incremented by 1. Now, every node will contain its depth.

C-7.5)

Algorithm indentedParentheticRepresentation(Tree T, Position v, int *indent*)

```
print out indent number of tabs
if T.isExternal(v) then
  print v.element().toString()
else
  indent++
  print v.element().toString() + "("
  Enumeration children_of_v = T.children(v)
  while children_of_v.hasMoreElements() do
    Position w = (Position) children_of_v.nextElement()
    indentedParentheticRepresentation(T, w, indent)
  end while
  indent--
  print out indent number of tabs
end if
```

C-7.6)

Let T_1 be a tree of $n/2$ nodes in a single path from the root to a single external node v. And let T_2 be a tree of $n/2$ nodes having $\lfloor n/2 \rfloor + 1$ external nodes. Now, create T by attaching T_2 to T_1 at v.

C-7.7)

We will show this using induction. For $n_I = 0$, then $n_E = 2n_I + 1 = 1$. This is obviously true. For $n_I = 1$, then $n_E = 2n_I + 1 = 2 + 1 = 3$. Again, this is obviously true from our problem definition. Now let us assume that the n_E equation holds true for $k' < k$, i.e., for any $n_I = k' < k$, $n_E = 2n_I + 1$.

Now consider $n_I = k$. Then, $n_E = 2(k-1) + 1 + (3-1)$. That is, the number of external nodes is equal to the number of external nodes for a tree with $k-1$ internal nodes plus 3 (we added an internal node which must have 3 children) minus 1 (in creating the new internal node, we made an external node into an internal node). Thus, $n_E = 2k - 2 + 3 = 2k + 1$. This is what we needed to show.

C-7.9)

One way to do this is the following: in the *external* method, set height and balance to be zero. Then, alter the *right* method as follows:

Algorithm right()
  if $v.isInternal(v)$ **then**
    if $v.leftchild.height > v.rightchild.height$ **then**
      $v.height = v.leftchild.height + 1$;
    else
      $v.height = v.rightchild.height + 1$;
    end if
    $v.balance = absval(v.rightchild.height - v.leftchild.height)$;
  end if
  printBalanceFactor(v)

C-7.12)

a) yes
b) no
c) yes, postorder.

C-7.21)

Algorithm LCA(Node v, Node w)
  int $v_{dpth} \leftarrow v.depth$
  int $w_{dpth} \leftarrow w.depth$
  while $v_{dpth} > w_{dpth}$ **do**
    $v \leftarrow v.parent$
  end while
  while $w_{dpth} > v_{dpth}$ **do**
    $w \leftarrow w.parent$
  end while
  while $v \neq w$ **do**
    $v \leftarrow v.parent$
    $w \leftarrow w.parent$
  end while
  return v

C-7.24)

With an array-based implementation, the binary tree methods positions() and iterator() take $O(n)$ time. This is because we need to step through the entire array and extract each position/element. The replace method takes constant time because of the fact that we can access elements in the array using an index, without having to search the entire array. The root, parent, children, leftChild, rightChild, and sibling methods are also constant because we can find these items with simple calculations (for example, the right child of a node with

index n is $n + 2$). Finally, isInternal, isExternal, and isRoot also only take a quick calculation based on indices (whether an index is in the last half of the array, is the first item in the array, etc.). Thus, these methods are also constant.

C-7.25)

Using a linked structure, the iterator() method is linear because we must still walk through the entire linked representation, getting every element. The root, parent, children, left, right, and sibling methods are all constant because we must follow only one or two links to determine these values. The replace method is also constant time because no traversal of the data structure is needed. We need only to change a few links/pointers in order to complete. The methods isInternal, isExternal, and isRoot are also constant since we need only to check local fields or links to determine these.

C-7.29)

```
Algorithm inorder(Tree T)
    Stack S ← new Stack()
    Node v ← T.root()
    push v
    while S is not empty do
      while v is internal do
        v ← v.left
        push v
      end while
      while S is not empty do
        pop v
        visit v
        if v is internal then
          v ← v.right
          push v
        end if
        while v is internal do
          v ← v.left
          push v
        end while
      end while
    end while
```