Problem Set 3-b Solution

Problem 1 For each node v in a tree T, let pre(v) be the rank of v in the preorder traversal of T, post(v) the rank of in the postorder traversal of T, depth(v) the depth of v, and desc(v) the number of descendants of v, not counting v itself. Derive a formula defining post(v) in terms of desc(v), depth(v), and pre(v), for each node v in T.

Solution: In the postorder traversal, when a node v is visited, all its descendants have been visited and all its ancestors have not been visited. On the other hand, in the preorder traversal, when a node v is visited, all its descendants have not been visited and all its ancestors have been visited. Therefore, we have post(v)= pre(v)+ desc(v)- depth(v).

Problem 2 Give an O(n)-time algorithm for computing the depths of all the nodes of a tree T, where n is the number of nodes of T.

Solution:

```
Algorithm computingDepths(v, d)
   {
     v.depth=d;
     if  v has no child
         return;
     else
        for  each child v' of v do
             computingDepths(v', d+1));
     return;
   }
```

To compute the depth of each node, simply call computeDepth(root, 0). This algorithm visits each node only once and each visit takes constant time. Therefore, it's running time is O(n).

Problem 3 Describe, in pseudo code, a nonrecursive algorithm for performing the preorder traversal on a binary tree in linear time.

Solution: We simulate the execution of the recursive algorithm for the preorder traversal by using a stack. In the recursive preorder traversal, we visit the root first, and recursively traverse the left subtree and then the right subtree. Initially, the stack contains the root only. Each time, we pop a node from the stack, visit it and push its right child and then the left child onto the stack. The algorithm is shown in pseudo code as follows:

```
Algorithm preOrder(v)
  {
    if  v = null
        return;
    Create an empty stack S;
    S.push(v);
    while S is not empty
```

```
        { v=S.pop();
          visit(v);
          if hasRight(v)
             S.push(right(v));
          if  hasLeft(v)
             S.push(left(v));
         }
  }
```

Time complexity analysis: Creating an empty stack takes O(1) time. The non-recursive algorithm pushes each node onto the stack once, pops it from the stack once and visits it once. Therefore, the algorithm takes O(n) time.

Problem 4 Describe, in pseudo code, a nonrecursive algorithm for performing the inorder traversal on a binary tree in linear time.

Solution: We simulate the execution of the recursive algorithm for the inorder traversal by using a stack. In the recursive inorder traversal, we recursively traverse the left subtree, visit the root, and then traverse the right subtree.

The non-recursive algorithm is show in pseudo code as follows:

```
    Algorithm inOrderTraversal(v)
       {
            Create an empty stack S;
            while S is not empty or v ≠ null
                if  v ≠ null   // keep going left until v=null
                  {  S.push(v);
                     v=left(v);
                   }
                else
                   {  v= S.pop();  // v has no left child, so visit it
                      visit(v);
                      v=right(v); // go to the right child
                    }
         }
```

Time complexity analysis: Creating an empty stack takes O(1) time. The non-recursive algorithm pushes each node onto the stack once, pops it from the stack once and visits it once. Therefore, the algorithm takes O(n) time.

Problem 5 Describe, in pseudo code, a nonrecursive algorithm for performing the postorder traversal on a binary tree in linear time.

Solution: We simulate the execution of the recursive algorithm for the postorder traversal by using a stack. In the recursive postorder traversal, we recursively traverse the left subtree, then traverse the right subtree, and lastly visit the root.

We introduce a variable named lastNodeVisited to keep track of the last node visited. The non-recursive algorithm is show in pseudo code as follows:

```
Algorithm postOrderTraversal(v)
    {
        Create an empty stack S;
        lastNodeVisited =null;
        while  S is not empty or v ≠ null
           if  v ≠ null   // push v and each left descendant onto the stack
             {   s.push(v);
                 v=left(v);
             }
           else
              { topNode=S.top();
                // S.top() returns the top node on the stack without removing it
                // if topNode has a right child not visited before,
                // then move to the right child of topNode
                if  topNode.right ≠ null and lastNodeVisited ≠ topNode.right
                    v=topNode.right;
                else // both left and right subtrees of topNode has been traversed
                    { visit(topNode);
                      lastNodeVisited = S.pop();
                    }
              }
       }
```

Time complexity analysis: Creating an empty stack takes O(1) time. The non-recursive algorithm pushes each node onto the stack once, pops it from the stack once peeks it (S.top()) once, and visits it once. Therefore, the algorithm takes O(n) time.

Problem 6 In the inorder traversal on a binary tree, a node v_i is the immediate inorder predecessor of a node v_j, if v_j is visited immediately after v_i in the inorder traversal. Describe a linear time algorithm for finding the immediate inorder predecessor of a node.

Solution: In the recursive inorder traversal, we recursively traverse the left subtree, visit the root, and then traverse the right subtree. We consider the following cases:

Case 1. v has a left child. If the left child of v does not have a right child, the left child of v is the IIP (Immediate Inorder Predecessor) of v. Otherwise, The rightmost descendant of the left child of v is the IIP of v.

Case 2. v does not have a left child. We further distinguish between the following three cases.

a. v is the right child of its parent. The IIP of v is its parent.
b. No ancestor of v is a right child. v has no IIP.
c. An ancestor of v is a right child. Find the first ancestor that is the right child of its parent, and the parent of this ancestor is the IIP.

The non-recursive algorithm is show in pseudo code as follows:

```
Algorithm inOrderPredecessor(v)
    {
      if v=null
          return null;
      if  left(v) != null   // v has a left child
        {
```

```
    v=left(v);  // the rightmost descendant of left(v) is the IIP
    while right(v)!=null
          v=right(v);
    return v;
   }
else
   {
    if v is root
       return null;
    while parent(v) != null
      {
        if  right(parent(v))=v  // v is the right child of its parent
            return parent (v);
        else
            v = parent(v);
       }
    return null ;  // no IIP
}
```

Time complexity analysis: only the nodes on the path between the immediate inorder predecessor and the node are visited and each visit takes O(1) time. Therefore, the algorithm takes O(h) time, where h is the height of the binary tree.

Problem 7 Let T be a binary tree. Define a **Roman node** to be a node v in T, such that the number of descendants in v's left subtree differs from the number of nodes in v's right subtree by at most 5. Describe a linear-time algorithm for finding each node v of T, such that v is not a Roman node, but all of v's descendants are Roman nodes.

Solution: The following solution is proposed by my student **Brian Price** of the 17s1 class. We use the recursive postorder traversal. For each node v, the algorithm returns a 2-tuple (m, l), where m is the size of the subtree rooted at v, and l is a linked list containing all the nodes in the subtree rooted at v each of which is not a Roman node, but all its descendants are Roman nodes.

```
Algorithm romanNode(v)
    {
      if !hasLeft(v) and !hasRight(v)
       {
         Create an empty list L;
       return (1, L);
      }
     else
      {
        if hasLeft(v)
          (N1, L1) = romanNode(left(v));
        else
          (N1, L1)=(0, null);
        if hasRight(v)
           (N2, L2) = romanNode(right(v));
        else (N2, L2)=(0, null);
        // visit this node
```

```
    if L1=null and L2=null and |N1 - N2| > 5:
      {
        // All the descendants of v are Roman nodes, so v is the first node
        // in the list
        Create empty list L;
        L.append(v);
        return (N1+N2+1, L);
      }
     // v is not a node in the list
    Concatenate L1 and L2 into a single linked list L;
    return (N1+N2+1, L);
  }
```

Time complexity analysis: this algorithm uses the postorder traversal, where each node is visited once and each visit takes constant time. Notice that concatenating two linked lists takes O(1) time by making the last element of one linked list point the first element of the other linked list. Therefore, the time complexity is O(n), where n is the number of nodes.

Problem 8 Let T be a binary tree with n nodes. Define the lowest common ancestor (LCA) of two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself). Given two nodes v and w, describe an efficient algorithm for finding the LCA of v and w. What is the running time of your algorithm?

Solution: Assume that each node has a visit flag that is initially 0.

```
Algorithm LCA(v, w)
  {
    x=v;
    /* Set the visit flag of each node on the path from v to the root */
    while ( x!=null )
      {
        x.visit=1;
        x=x.parent;
      }
    x=w;
    while ( x.visit=0 )  // search for the LCA
      x=x.parent;
    y=v;
    while ( y!=null)  // Reset the visit flag
      {
        y.visit=0;
        y=y.parent;
       }
    return x // x is the LCA of v and w

  }
```

The first while loop takes O(depth(v)) time. The second while loop takes O(depth(w)) time, and the third while loop takes O(depth(v)) time. Therefore, this algorithm takes O(h) time, where h is the height of the tree.

This problem can also be solved in O(n) time by using the postorder traversal, where n is the number of nodes in the tree.

Additional notes: The fastest algorithm for this problem was proposed by Harel and Tarjan [1]. Their algorithm takes constant time after a preprocessing that takes O(n) time, where n is the number of the nodes in the tree.

Refereneces:
[1] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing, 13(2): 338355, 1984.