1. Show how to use an Euler tour traversal to compute the level number $f(p)$, as defined in Section 8.3.2, of each position in a binary tree $T$.

**Algorithm.**

The level number of the root is initialized with 0. The algorithm progresses like a standard Euler traversal starting at the root. The key is passing the level number of a node around as a parameter to the traversal. When descending to the left, we pass $2l + 1$, and when we recurse on the right, we pass $2l + 2$. Then, during the first visit we assign the nodes level numbering. The pseudocode of the non-recursive helper method is shown in Algorithm 1 and the main recursive algorithm is shown in Algorithm 2.

**Algorithm 1 AssignLevelNumbering($T$)**

Input: Binary tree $T$

1: AssignLevelNumbering($T, T.root()$, 0)

**Algorithm 2 AssignLevelNumbering($T, p, l$)**

Input: Binary tree $T$, position $p$ and level numbering $l$

1: $T.setLevelNumbering(p, l)$
2: if $T.isInternal(p)$ then
3: AssignLevelNumbering($T.left(p), 2l + 1$)
4: AssignLevelNumbering($T.right(p), 2l + 2$)

**Time Complexity.**

**Theorem 1.** Algorithm 2 runs in $O(n)$ time on a proper binary tree with $n$ nodes.

*Proof. Using a linked structure to implement the binary tree, left(), right(), and isInternal() run in $O(1)$ time. Because each node is visited three times in an Euler traversal, and each of the visit actions clearly take $O(1)$ time. Visiting $n$ nodes takes $3O(n) = O(n)$ time. 

**Memory Complexity.**

**Theorem 2.** Algorithm 2 runs in $O(h)$ additional memory on a proper binary tree of height $h$.

*Proof. The traversal is implemented recursively. Each function call incurs a constant memory overhead to hold the values of the parameters. The only active function calls in the stack will be a node and all of its ancestors, thus the total memory overhead is $O(h)$. 

2. Describe, in pseudocode, a nonrecursive method for performing an inorder traversal of a binary tree in linear time.

**Algorithm.**

The algorithm is shown in Algorithm 3. Essentially, instead of using the function stack with recursion, we use an explicit stack to store positions of the tree. In each iteration, we attempt to go left as far as possible. When we can no longer go left, we pop from the stack, visit, and go right one step. Doing this traverses the tree in an inorder traversal.

<table>
<thead>
<tr>
<th>Algorithm 3 Nonrecursive inorder traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Binary tree $T$</td>
</tr>
<tr>
<td>1: Stack of positions $S \leftarrow \emptyset$</td>
</tr>
<tr>
<td>2: Position $c \leftarrow T.root()$</td>
</tr>
<tr>
<td>3: while $\neg S.isEmpty() \lor c \neq \emptyset$ do</td>
</tr>
<tr>
<td>4: if $c \neq \emptyset$ then</td>
</tr>
<tr>
<td>5: $S.push(c)$</td>
</tr>
<tr>
<td>6: $c \leftarrow T.left(c)$</td>
</tr>
<tr>
<td>7: else</td>
</tr>
<tr>
<td>8: $c \leftarrow S.pop()$</td>
</tr>
<tr>
<td>9: visit($c$)</td>
</tr>
<tr>
<td>10: $c \leftarrow T.right(c)$</td>
</tr>
</tbody>
</table>

**Time Complexity.**

**Theorem 3.** Algorithm 3 runs in $O(n)$ time on a proper binary tree with $n$ nodes.

*Proof.* Using a linked structure to implement the binary tree and an array to implement the stack, the tree and stack ADT functions run in $O(1)$ time. Since each node is pushed on the stack, popped from the stack, and visited once the loop runs in time linear to the number of nodes. Assuming the visit action is constant time, the total cost of the algorithm is thus $O(n)$. □

**Memory Complexity.**

**Theorem 4.** Algorithm 3 runs in $O(h)$ additional memory on a proper binary tree of height $h$.

*Proof.* Using a linked structure to implement the binary tree and an array to implement the stack, the maximum height of the stack is proportional to the height of the tree. Thus the memory complexity is $O(h)$. □
3. Let $T$ be a tree with $n$ positions. Define the lowest common ancestor (LCA) between two positions $p$ and $q$ as the lowest position in $T$ that has both $p$ and $q$ as descendants (where we allow a position to be a descendant of itself). Given two positions $p$ and $q$, describe and analyze an efficient algorithm for finding the LCA of $p$ and $q$.

**Algorithm.**

Essentially, the idea is to create a stack of all ancestors for $p$ and $q$ using a stack. Then, pop all elements of the stacks to find the LCA of each node. The code is shown in Algorithm 4.

---

**Algorithm 4 Least Common Ancestor**

**Input:** Binary tree $T$, positions $p$ and $q$

**Output:** Least common ancestor $l$

1. Stacks $S_p \leftarrow \emptyset$ and $S_q \leftarrow \emptyset$
2. Position $n \leftarrow p$
3. while $n \neq \emptyset$ do
   4. $S_p$.push($n$)
   5. $n \leftarrow T$.parent($n$)
   6. $n \leftarrow q$
4. while $n \neq \emptyset$ do
   8. $S_q$.push($n$)
   9. $n \leftarrow T$.parent($n$)
4. Position $l = \emptyset$
11. while $S_p$.top() = $S_q$.top() do
   12. $l \leftarrow S_p$.top()
   13. $S_p$.pop()
   14. $S_q$.pop()
15. return $l$

---

**Time Complexity.**

**Theorem 5.** Algorithm 4 runs in time $O(d_p + d_q)$, where $d_p$ is the depth of position $p$ and $d_q$ is the depth of position $q$.

**Proof.** Given that the tree is implemented with a linked structure and the stacks are implemented with arrays, all of the tree and stack ADT functions run in $O(1)$ time. The first while loop clearly iterates $d_p$ times and the second while loop clearly iterates $d_q$ times. Since the stacks now store $d_p$ and $d_q$ nodes each, the final while loop could execute at most $\min(d_q, d_p)$. Thus in total this algorithm runs in $O(d_p + d_q)$ time.

---

**Memory Complexity.**

**Theorem 6.** Algorithm 4 uses $O(d_p + d_q)$ additional memory, where $d_p$ is the depth of position $p$ and $d_q$ is the depth of position $q$.

**Proof.** Clearly after the first two loops execute the stacks accumulate $d_p$ and $d_q$ positions respectively. The final loop only removes items from the stacks. So in total the algorithm uses $O(d_p + d_q)$ additional memory.
4. **Bonus.** Two ordered trees $T'$ and $T''$ are said to be isomorphic if one of the following holds:

- Both $T'$ and $T''$ are empty.
- Both $T'$ and $T''$ consist of a single node.
- The roots of $T'$ and $T''$ have the same number $k \geq 1$ of subtrees, and the $i^{th}$ such subtree of $T'$ is isomorphic to the $i^{th}$ such subtree of $T''$ for $i = 1, \ldots, k$.

Design and analyze an algorithm that tests whether two given ordered trees are isomorphic.

**Algorithm.**

Algorithm 5 computes whether two trees are isomorphic with a preorder traversal. It is initially called with the roots of the two trees. Then, it checks that each node has the same number of children, and then each of the subtrees are also isomorphic with recursion.

**Algorithm 5 Isomorphic($T', T''$)**

**Input:** Trees $T'$ and $T''$

**Output:** Boolean value of isomorphic nature of $T'$ and $T''$

1. return Isomorphic($T', T'.root(), T'', T''.root()$)

**Algorithm 6 Isomorphic($T', n', T'', n''$)**

**Input:** Tree $T'$, position $n'$ from $T'$, tree $T''$, and position $n''$ from $T''$

**Output:** Boolean value of isomorphic nature of $T'$ and $T''$

1. if $T'.numChildren(n') \neq T''.numChildren(n'')$ then
2. return false
3. for $i \leftarrow 1$ to $T'.numChildren(n')$ do
4. if $\neg$Isomorphic($T', T'.getIthChild(n', i), T''.getIthChild(n'', i)$) then
5. return false
6. return true

**Time Complexity.**

**Theorem 7.** Algorithm 5 runs in $O(\min(n, m))$ time, where $n$ is the size of $T'$ and $m$ is the size of $T''$.

**Proof.** Assuming a linked structure of the tree, all of the ADT functions will run in $O(1)$ time. Since the algorithm follows a preorder traversal, the algorithm runs in $O(\min(n, m))$ time. It will clearly only execute at most the number of elements in the smallest tree.

**Memory Complexity.**

**Theorem 8.** Algorithm 5 runs in $O(\min(h_{T'}, h_{T''}))$ where $h$ is the height of the respective tree.

**Proof.** The depth of the recursion cannot be more then the height of the smaller tree clearly. Thus the additional memory overhead of the recursion is $O(\min(h_{T'}, h_{T''}))$. 

4