1. Provide statements and proofs for the following:

(a) What is the minimum number of internal nodes for an improper binary tree with \( n \) nodes?
(b) What is the maximum number of internal nodes for an improper binary tree with \( n \) nodes?
(c) What is the minimum number of external nodes for an improper binary tree with \( n \) nodes?
(d) What is the maximum number of external nodes for an improper binary tree with \( n \) nodes?
(e) Let \( T \) be a proper binary tree with height \( h \) and \( n \) nodes. Show that \( \log(n+1) - 1 \leq h \leq (n-1)/2 \).
(f) For the prior question what structure of trees yield the upper and lower bounds?

Solutions.

(a) \textbf{Theorem 1.} An improper binary tree of \( n \) nodes has at a minimum \( \frac{n-1}{2} \) internal nodes.

\textit{Proof.} The fewest number of internal nodes occurs when a binary tree has minimum height. To prove this, assume that a binary tree could have fewer than \( \frac{n-1}{2} \) internal nodes, say \( \frac{n-1}{2} - 1 \) internal nodes. However, we know that each internal node can have at most two children \( 2\left(\frac{n-1}{2} - 1\right) \) and in total must support \( n-1 \) node (root has no parent). Clearly, \( 2\left(\frac{n-1}{2} - 1\right) = n-3 < n-1 \), implying by the pigeon hole principle that some internal node has more than three children breaking our assumptions. Thus by contradiction, the minimum number of internal nodes will be \( \frac{n-1}{2} \). \qed

(b) \textbf{Theorem 2.} An improper binary tree of \( n \) nodes has at a maximum \( n-1 \) internal nodes.

\textit{Proof.} The maximum number of internal nodes is essentially a linked-list structure. Every node has exactly one child except the last which is external. It is impossible to have \( n \) internal nodes, because at least one cannot have a child. \qed

(c) \textbf{Theorem 3.} An improper binary tree of \( n \) nodes has at a minimum 1 external nodes.

\textit{Proof.} Same logic as found in the proof of Theorem 2. \qed

(d) \textbf{Theorem 4.} An improper binary tree of \( n \) nodes has at a maximum \( \frac{n+1}{2} \) external nodes.

\textit{Proof.} Same logic as found in the proof of Theorem \[1\] \qed
Theorem 5. In a proper binary tree with height $h$ and $n$ nodes, $\log(n + 1) - 1 \leq h \leq \frac{n-1}{2}$.

Proof. To prove this we separate into two parts.

i. First, we show $\log(n + 1) - 1 \leq h$ using induction.
   Base case: For a tree of size $n = 1$, it is trivially true.
   Inductive step: Assume $\log(n + 1) - 1 \leq h$ for a tree of size $n' = n + 2$ (with height $h'$). So, in a tree of size $n'$ either $h' = h$ or $h' = h + 1$. In the first case, the last level of the tree either did not fill up entirely, or it became full. In either case, $\log(n + 2 + 1) - 1 < h' = h$ (the logarithm does not increment by 1!). In the second case, the height increases by one, clearly $\log(n + 2 + 1) - 1 \leq h + 1$.

ii. Second, we show $h \leq \frac{n+1}{2}$ using induction.
   Base case: For a tree of size $n = 1$, it is trivially true.
   Inductive step: Assume $h \leq \frac{n-1}{2}$ for a tree of size $n$. We must show that $h' \leq \frac{n'+1}{2}$ holds for a tree of size $n' = n + 2$ (with height $h'$). So in a tree of size $n' = n + 2$ either $h' = h$ or $h' = h + 1$. The first case is trivially true, $h' = h \leq \frac{n-1}{2} < \frac{n+1}{2}$. In the second case, $h' = h + 1 \leq \frac{n+1}{2} = \frac{n-1}{2} + 1$, thus the inequality still holds.

(f) The upper bound is a tree with a linear structure maximizing the height. Each node will have at least one external node. The lower bound is a complete binary tree willing every level to its maximum capacity thus minimizing the height.

2. 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 is shown in Algorithm 1.

Algorithm 1 AssignLevelNumbering()

<table>
<thead>
<tr>
<th>Input:</th>
<th>Node $n$ and level numbering $l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>$n$.levelNumbering ← $l$</td>
</tr>
<tr>
<td>2:</td>
<td>if $n$.isInternal() then</td>
</tr>
<tr>
<td>3:</td>
<td>AssignLevelNumbering($n$.left(), $2l + 1$)</td>
</tr>
<tr>
<td>4:</td>
<td>if $n$.isInternal() then</td>
</tr>
<tr>
<td>5:</td>
<td>AssignLevelNumbering($n$.right(), $2l + 2$)</td>
</tr>
</tbody>
</table>

Time Complexity.

Theorem 6. Algorithm 1 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 7. Algorithm 1 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 $O(h)$.

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 2.

Algorithm 2 Least Common Ancestor

Input: Nodes $p$ and $q$
Output: Least common ancestor $l$
1: Stacks $S_p \leftarrow \emptyset$ and $S_q \leftarrow \emptyset$
2: Node $n \leftarrow p$
3: while $n \neq \emptyset$ do
4: $S_p$.push($n$)
5: $n \leftarrow n$.parent()
6: $n \leftarrow q$
7: while $n \neq \emptyset$ do
8: $S_q$.push($n$)
9: $n \leftarrow n$.parent()
10: Node $l = \emptyset$
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 8. Algorithm 2 runs in time $O(d_p + d_q)$, where $d_p$ is the depth of node $p$ and $d_q$ is the depth of node $q$.

Proof. Given that the tree is implemented with a linked structure and the stacks are implemented with arrays, all of the node 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 9. Algorithm 2 uses $O(d_p + d_q)$ additional memory, where $d_p$ is the depth of node $p$ and $d_q$ is the depth of node $q$. 


Proof. Clearly after the first two loops execute the stacks accumulate \( d_p \) and \( d_q \) nodes 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 3 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 3 Isomorphic()**

*Input:* Nodes \( n' \) from \( T' \) and \( n'' \) from \( T'' \)

*Output:* Boolean value of isomorphic nature of \( T' \) and \( T'' \)

1. if \( n'.\text{numChildren}() \neq n''.\text{numChildren}() \) then
   2. return false
3. \( k \leftarrow n'.\text{numChildren}() \)
4. for \( i \leftarrow 1, \ldots, k \) do
5.   if \( \neg\text{Isomorphic}(n'.\text{getIthChild}(i), n''.\text{getIthChild}(i)) \) then
6.     return false
7. return true

**Time Complexity.**

Theorem 10. Algorithm 3 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 11. Algorithm 3 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 \( O(\min(h_{T'}, h_{T''})) \).