Homework 5

OPTIONAL PROBLEMS
(No due date)

1 Written Problems

First Common Ancestor

Design an algorithm and write code to find the first (farthest from the root) common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure.
Note: This is not necessarily a binary *search* tree.

Solution:

```
/* We know we have found the first (farthest from the root)
common ancestor when p and q are not "related" on the same
side of the tree from a given ancestor (curr)  */

commonAncestor(Node curr, Node p, Node q):
         """commonAncestor: root node (curr) and p, q nodes -> node
         Purpose: Find the first ancestor of the p and q nodes in the tree
         """
     if (related(curr.left, p) && related(curr.left, q))
          return commonAncestor(curr.left, p, q)
     if (related(curr.right, p) && related(curr.right, q))
          return commonAncestor(curr.right, p, q)
     return curr

/* Determines if two inputted nodes are related.
Traverses down the tree from ancestor until the
descendant is found (returns true) or there are no
more nodes to be checked (returns false) */

related(Node ancestor, Node descendant):
     if (ancestor == null)
        return false
     if (ancestor == descendant)
        return true
     return related(ancestor.left, descendant) || related(ancestor.right, descendant)
```

Next Node

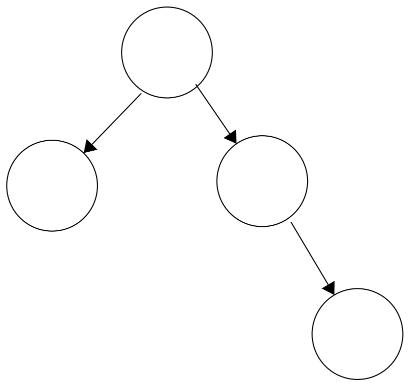
Write an algorithm to find the 'next' node (e.g., in-order successor) of a given node in a binary search tree where each node has a link to its parent.

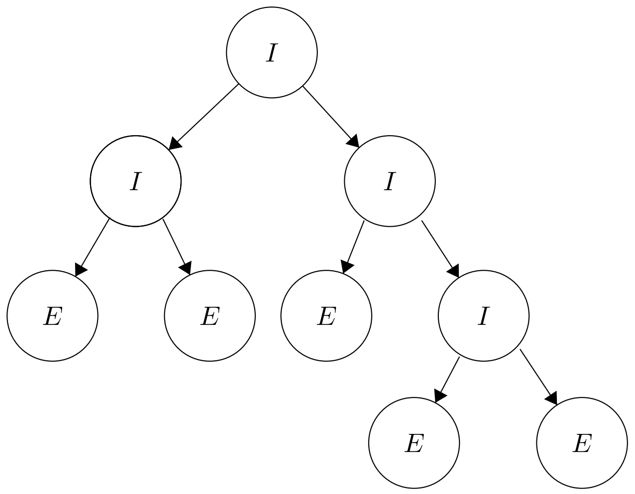
Solution:

```
findNext(Node n):
     """findNext: node -> node
     Purpose: find the n's 'next' node in a binary search tree
     """
     if (n != null):
          Node p
          if (n.parent == null || n.right != null):
               p = leftMostChild(n.right)
          else:
               while ((p = n.parent) != null):
                    if (p.left == n):
                         break
                    n = p
          return p

     return null;

leftMostChild(Node n):
     if (n == null) return null
     while (n.left != null):
        n = n.left
     return n
```

External Nodes

An extended binary tree is a tree, such as the one pictured below, in which each missing child is replaced with an external node (labled with an E). Prove by strong induction that an extended binary tree with n internal nodes has n+1 external nodes.



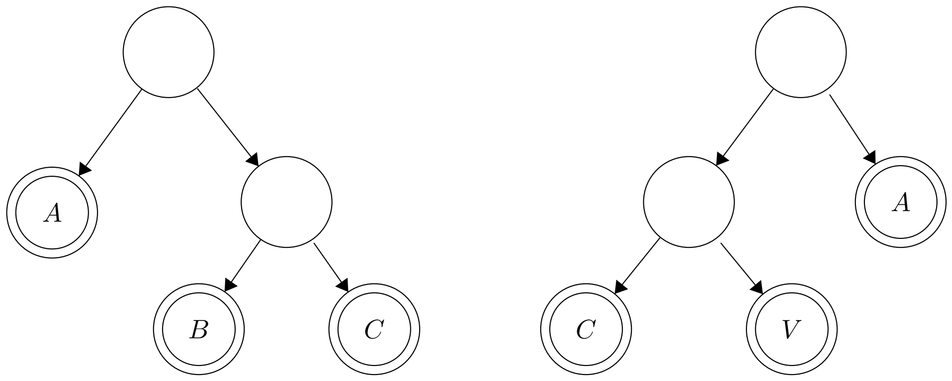


Solution:

Proof. By induction on n the number of internal nodes in a tree. Let P(n) represent the statement that "for all such binary trees with n internal nodes, the number of leaves is always one more than the number of internal nodes".
Base case: P(0) is the case in which there is only one node in a tree necessitating that it is a leaf. P(0)= 1 = n + 1.
Induction step: for n $\geq$ 1 and k = n+1, prove P (n) implies P (k)
Goal: for n $\geq$ 1, for all k < n, P(k) implies P(n). Given a binary tree with n $\geq$ 1 internal nodes, we know it has two subtrees with numbers of internal nodes given by l and r such that l + r + 1 = n (all of the internal nodes of the given tree are distributed between the left subtree, the right subtree, and the root). By our inductive hypothesis, P (l) and P (r) tell us that the number of leaves in the left and right subtrees (which are smaller than the given tree) are l + 1 and r + 1 respectively. The total number of leaves in the given tree is then their sum l + r + 2. By our definition, the number of internal nodes in the given tree is l + r + 1. The total number of leaves is (l + r + 1) + 1, exactly one more than the number of internal nodes. Thus P (n) holds.
By induction, the original claim is proven for any tree of the given structure.

Reverse Leaves

Write an algorithm to reverse the order of the leaves in a given binary tree. The tree can be manipulated and changed, so long as the leaves are reversed. For example:



Solution:

```
reverseLeaves(Node curr):
        """reverseLeaves: root node (curr) -> root node
        Purpose: reverse the leaves of the tree and return its root
        """

    //if curr in null, return null
    if (curr == null):
        return null

    //if curr has a left and right child, switch them
    if (curr.left != null && curr.right != null):
        temp = curr.left
        curr.left = curr.right
        curr.right = temp

    //reverse the leaves of the subtrees with the left and right child as roots
    reverseLeaves(curr.left)
    reverseLeaves(curr.right)

    //the leaves have been reversed since the whole tree is reversed, so return the root
    return curr
```