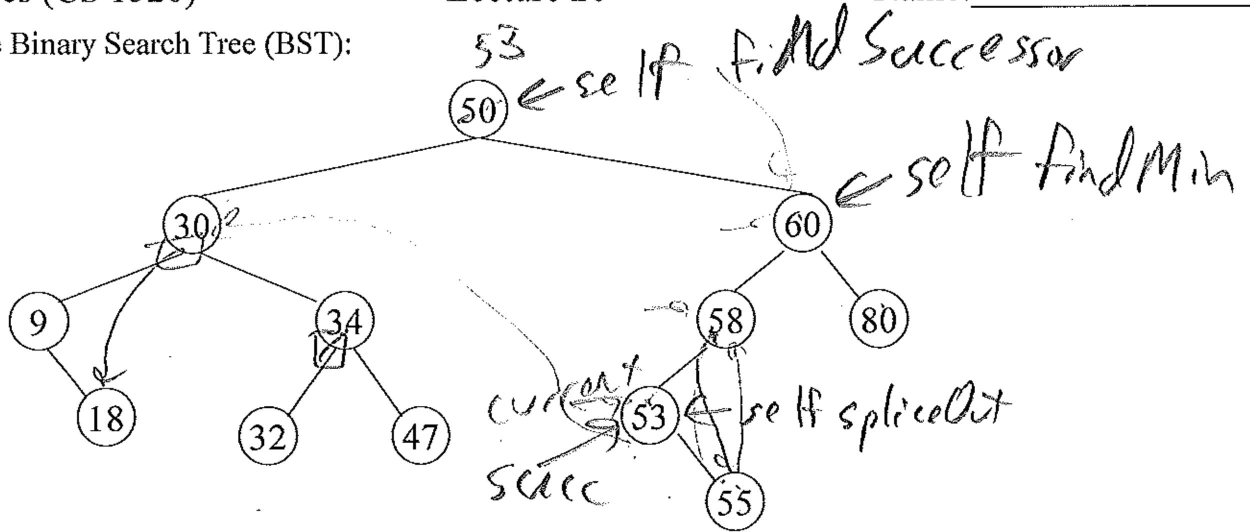
Data Structures (CS 1520) Lecture 20 Name:_______________

1. Consider the Binary Search Tree (BST):



a. What would need to be done to delete 32 from the BST? change its parent's leftChild to None

b. What would need to be done to delete 9 from the BST?
Change its parent's leftChild pointer to 9's rightChild

c. What would be the result of deleting 50 from the BST? Hint: One technique when programming is to convert a hard problem into a simpler problem. Deleting a BST node that contains two children is a hard problem. Since we know how to delete a BST node with none or one child, we can convert "deleting a node with two children" problem into a simpler problem by overwriting 50 with another node's value. Which nodes can be used to overwrite 50 and still maintain the BST ordering? either largest value from left subtree (47), or smallest value from right subtree (53)

d. Which node would the `TreeNode's findSuccessor` method return for `succ` if we are deleting 50 from the BST?
(53)

2. When the `findSuccessor` method is called how many children does the `self` node have? two children

3. How could we improve the `findSuccessor` method? - eliminate "dead code" that cannot execute.

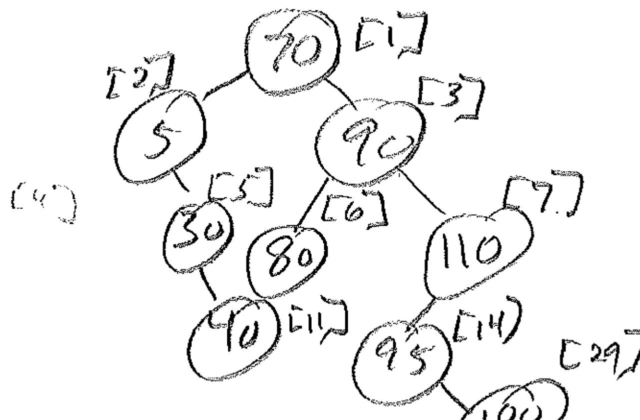
4. When the `spliceOut` method is called from `remove` how many children could the `self` node have?
at most a right child

5. How could we improve the `spliceOut` method?
eliminate "dead code"

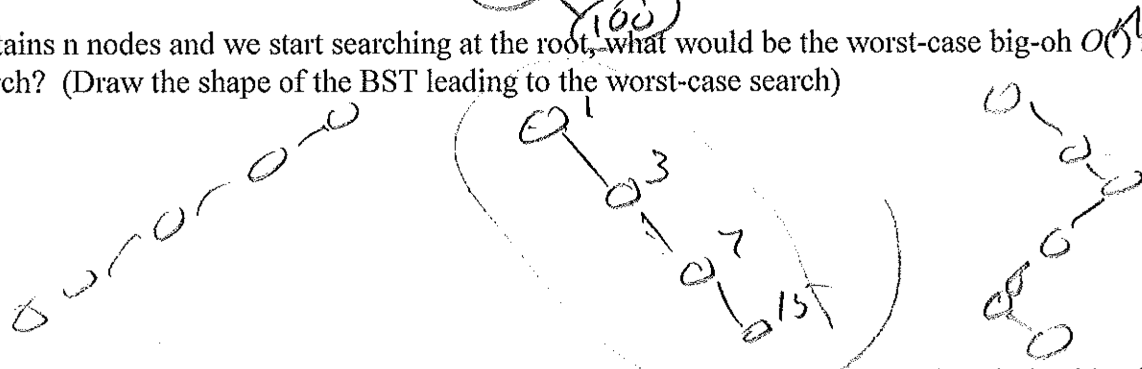
6. The shape of a BST depends on the order in which values are added (and deleted).

a) What would be the shape of a BST if we start with an empty BST and insert the sequence of values:

70, 90, 80, 5, 30, 110, 95, 40, 100



b) If a BST contains n nodes and we start searching at the root, what would be the worst-case big-oh $O()$ notation for a successful search? (Draw the shape of the BST leading to the worst-case search)



7. We could store a BST in an array like we did for a binary heap, e.g. root at index 1, node at index i having left child at index 2 * i, and right child at index 2 * i + 1.

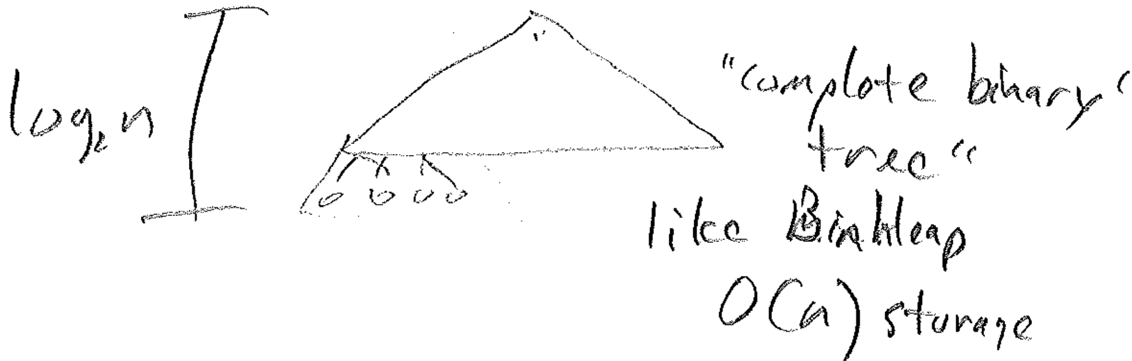
a) Draw the above BST (after inserting: 70, 90, 80, 5, 30, 110, 95, 40, 100) stored in an array (leave blank unused slots)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	29
Index 0 Not Used	70	5	90		30	80	110				40			95								100

b) What would be the worst-case storage needed for a BST with n nodes?

Storage $O(2^n)$ slots in Python list

8. a) If a BST contains n nodes, draw the shape of the BST leading to best, successful search in the worst case.



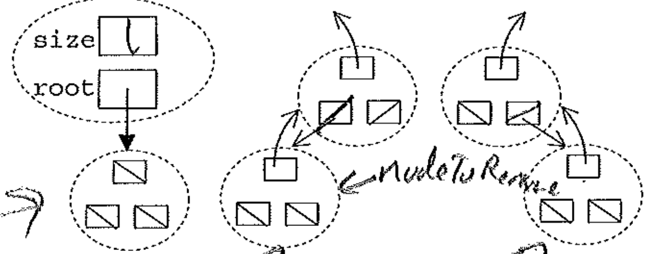
b) What is the worst-case big-oh $O()$ notation for a successful search in this "best" shape BST?

$O(\log_2 n)$

2. More partial TreeNode class and partial BinarySearchTree class.

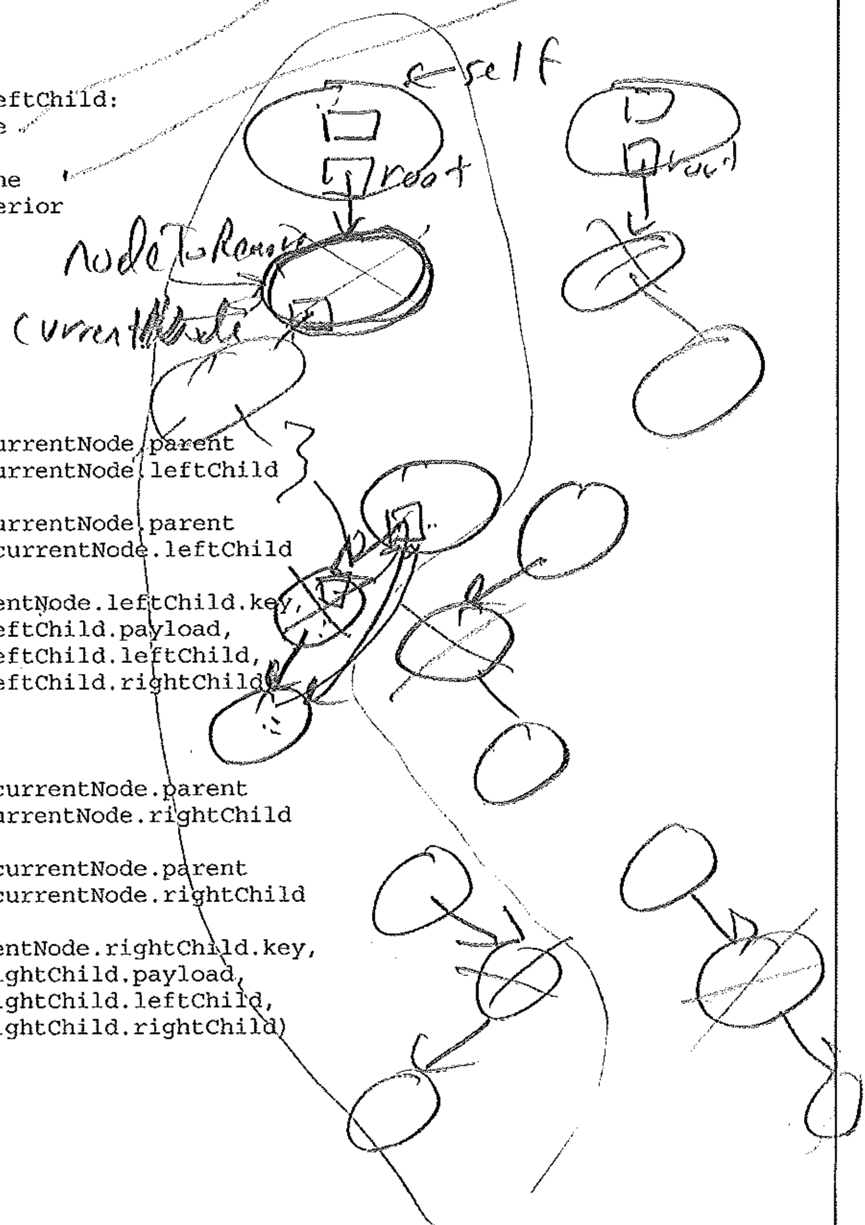
```
class BinarySearchTree:
   . . .
  def delete(self,key):
    if self.size > 1:
        nodeToRemove = self._get(key,self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size = self.size-1
        else:
            raise KeyError('Error, key not in tree')
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size = self.size - 1
    else:
        raise KeyError('Error, key not in tree')

  def __delitem__(self,key):
      self.delete(key)


  def remove(self,currentNode):
    if currentNode.isLeaf(): #leaf
      if currentNode == currentNode.parent.leftChild:
          currentNode.parent.leftChild = None
      else:
          currentNode.parent.rightChild = None
    elif currentNode.hasBothChildren(): #interior
      succ = currentNode.findSuccessor()
      succ.spliceOut()
      currentNode.key = succ.key
      currentNode.payload = succ.payload

    else: # this node has one child
      if currentNode.hasLeftChild():
        if currentNode.isLeftChild():
            currentNode.leftChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.leftChild
        elif currentNode.isRightChild():
            currentNode.leftChild.parent = currentNode.parent
            currentNode.parent.rightChild = currentNode.leftChild
        else:
            currentNode.replaceNodeData(currentNode.leftChild.key,
                                   currentNode.leftChild.payload,
                                   currentNode.leftChild.leftChild,
                                   currentNode.leftChild.rightChild)

      else:
        if currentNode.isLeftChild():
            currentNode.rightChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.rightChild
        elif currentNode.isRightChild():
            currentNode.rightChild.parent = currentNode.parent
            currentNode.parent.rightChild = currentNode.rightChild
        else:
            currentNode.replaceNodeData(currentNode.rightChild.key,
                                   currentNode.rightChild.payload,
                                   currentNode.rightChild.leftChild,
                                   currentNode.rightChild.rightChild)
```

a) Update picture where we delete a leaf.



b) Where in the code is each handled?

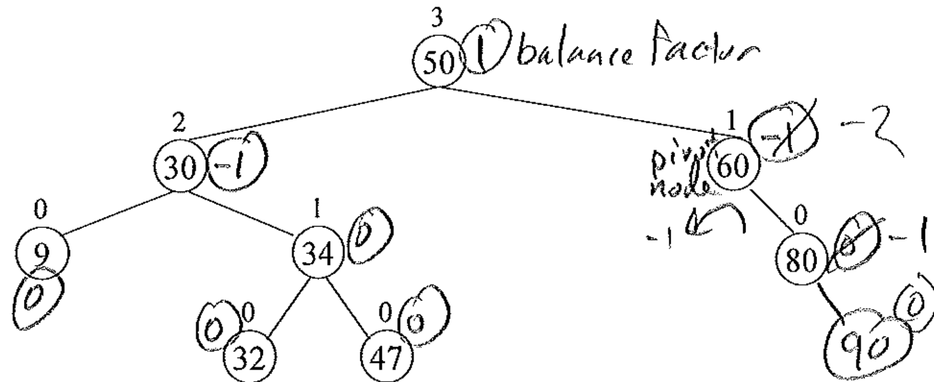
c) Draw all pictures deleting all nodes with one child.



3. Yet even more partial TreeNode class and partial BinarySearchTree class.

```
class TreeNode:
   . . .
      def findSuccessor(self):
         succ = None
         if self.hasRightChild():
             succ = self.rightChild.findMin()
         else:
             if self.parent:
                 if self.isLeftChild():
                     succ = self.parent
                 else:
                     self.parent.rightChild = None
                     succ = self.parent.findSuccessor()
                     self.parent.rightChild = self
         return succ

      def findMin(self):
         current = self
         while current.hasLeftChild():
             current = current.leftChild
         return current

      def spliceOut(self):
         if self.isLeaf():
            if self.isLeftChild():
               self.parent.leftChild = None
            else:
               self.parent.rightChild = None
         elif self.hasAnyChildren():
            if self.hasLeftChild():
               if self.isLeftChild():
                  self.parent.leftChild = self.leftChild
               else:
                  self.parent.rightChild = self.leftChild
               self.leftChild.parent = self.parent
            else:
               if self.isLeftChild():
                  self.parent.leftChild = self.rightChild
               else:
                  self.parent.rightChild = self.rightChild
               self.rightChild.parent = self.parent
```

1. An *AVL Tree* is a special type of Binary Search Tree (BST) that it is *height balanced.* By height balanced I mean that the height of every node's left and right subtrees differ by at most one. This is enough to guarantee that a AVL tree with n nodes has a height no worst than $O(1.44 \log_2 n)$. Therefore, insertions, deletions, and search are worst case $O(\log_2 n)$. An example of an AVL tree with integer keys is shown below. The height of each node is shown.



Each AVL-tree node usually stores a *balance factor* in addition to its key and payload. The balance factor keeps track of the relative height difference between its left and right subtrees, i.e., height(left subtree) - height(right subtree).

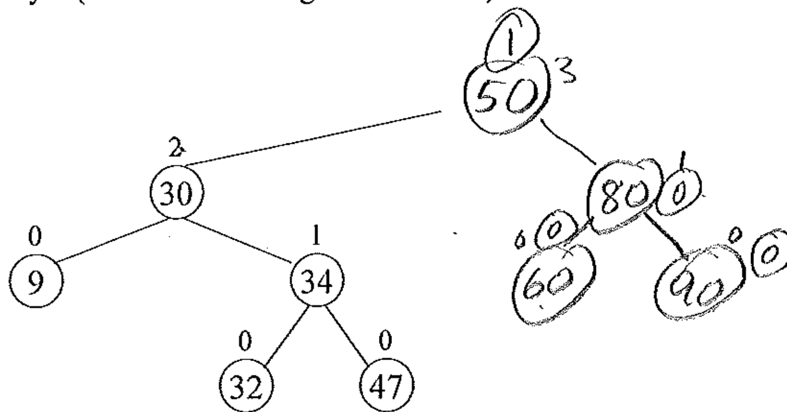
a) Label each node in the above AVL tree with one of the following *balance factors*:

- 0 if its left and right subtrees are the same height
- 1 if its left subtree is one taller than its right subtree
- -1 if its right subtree is one taller than its left subtree

b) We start a `put` operation by adding the new item into the AVL as a leaf just like we did for Binary Search Trees (BSTs). Add the key 90 to the above tree.

c) Identify the node "closest up the tree" from the inserted node (90) that no longer satisfies the height-balanced property of an AVL tree. This node is called the *pivot node.* Label the pivot node above. (60)

d) Consider the subtree whose root is the pivot node. How could we rearrange this subtree to restore the AVL height balanced property? (Draw the rearranged tree below)



2. Typically, the addition of a new key into an AVL requires the following steps:

- compare the new key with the current tree node's key (as we did in the `_put` function called by the `put` method in the BST) to determine whether to recursively add the new key into the left or right subtree
- add the new key as a leaf as the base case(s) to the recursion
- recursively (`updateBalance` method) adjust the balance factors of the nodes on the search path from the new node back up toward the root of the tree. If we encounter a pivot node (as in question (c) above) we perform one or two "rotations" to restore the AVL tree's height-balanced property.

For example, consider the previous example of adding 90 to the AVL tree. Before the addition, the pivot node (60) was already -1 ("tall right" - right subtree had a height one greater than its left subtree). After inserting 90, the pivot's right subtree had a height 2 more than its left subtree (balance factor -2) which violates the AVL tree's height-balance property. This problem is handled with a *left rotation* about the pivot as shown in the following generalized diagram:

Before the addition:

from parent

B -1; D 0; T_A height n - 1; T_C height n - 1; T_E height n - 1

After the addition, but before rotation:

from parent

B -2; D -1; T_A height n - 1; T_C height n - 1; T_E height n; new node

Rotate Left at Pivot

Recursive updateBalance method finds the pivot and calls the rebalance method to perform proper rotation(s)

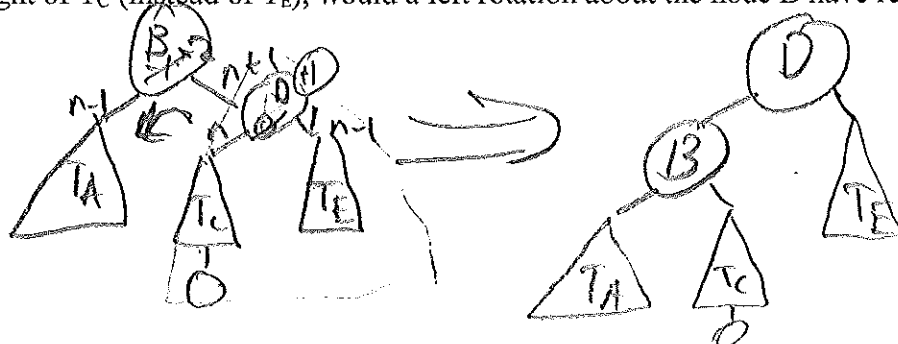
(D's balance factor was already adjusted before the pivot is found by the recursive updateBalance method which moves toward the root)

After left rotation at pivot:

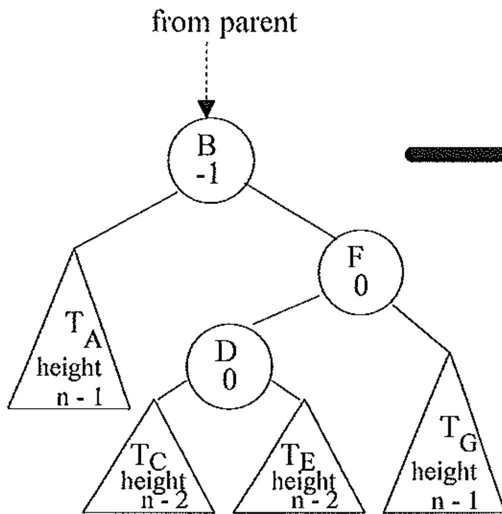
from parent

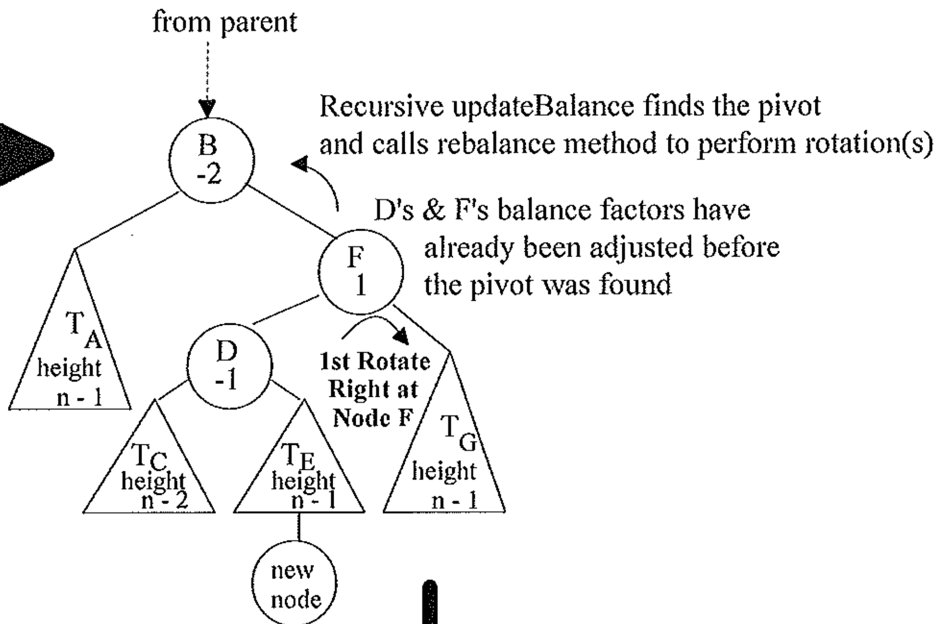
D 0; B 0; T_A height n - 1; T_C height n - 1; T_E height n; new node

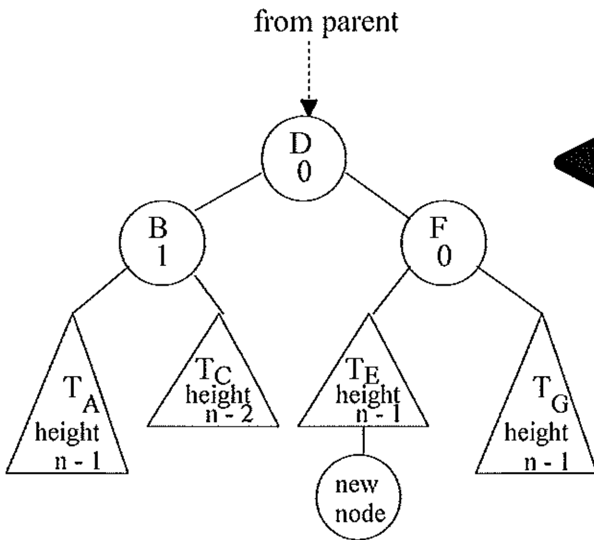
a) Assuming the same initial AVL tree (upper, left-hand of above diagram) if the new node would have increased the height of T_C (instead of T_E), would a left rotation about the node B have rebalanced the AVL tree? No

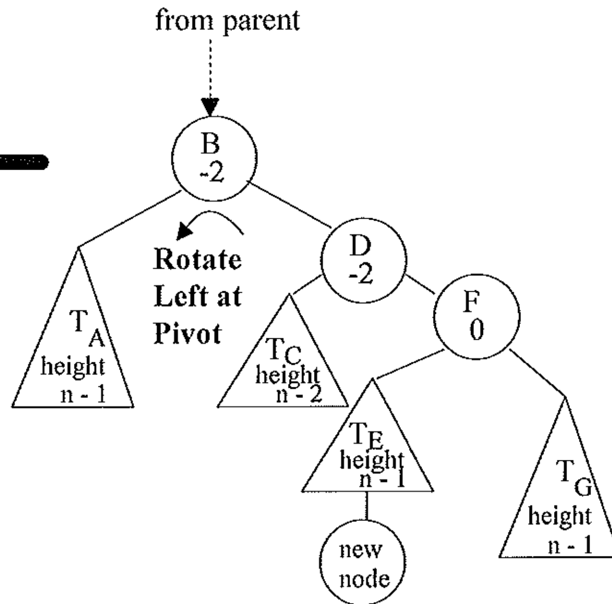


b) Before the addition, if the pivot node was already -1 (tall right) and if the new node is inserted into the left subtree of the pivot node's right child, then we must do two rotations to restore the AVL-tree's height-balance property.









b) Suppose that the new node was added in T_C instead of T_E, then the same two rotations would restore the AVL-tree's height-balance property. However, what should the balance factors of nodes B, D, and F be after the rotations?