CMSC 420: Fall 2022

Solutions to Homework 2: Search Trees

Solution 1:

(a) See Fig. 1. (Balance factors are shown in blue, where "`-`" means -1 and "`+`" means $+1$.) We have not included the heights, but they are part of the structure (at least for the implementation given in class) and are used to compute balance factors.

We first insert a new node 5 as the left child of 6. We then retrace the search path back to the root, updating balance factors as we go. This decreases the balance factors at 6 and 7 to -1, increases the balance factor at 4 to $+1$, and decreases the balance factor at 9 to -2, which is invalid. We find that 9 is left-right heavy, that is, its height is determined by its left-right grandchild, 7. This induces a left-right double rotation at 9, which performs a left rotation at 4 followed by a right rotation at 9. This brings 7 up to the root. We update the balance factors of the affected nodes (4, 7, and 9). Node 7 is the new root of the tree, and we are done.

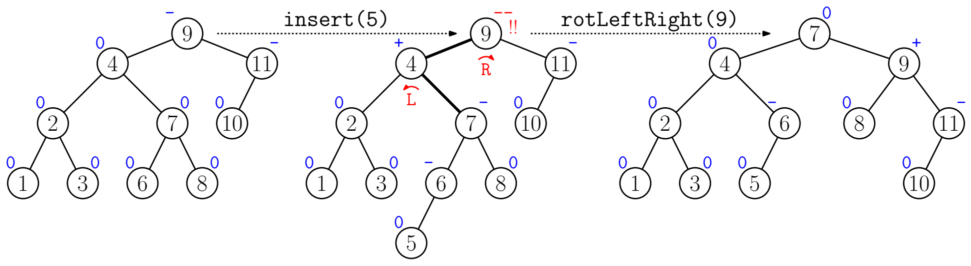

Figure 1: AVL tree: `insert(5)`. (Node heights not shown.)

(b) See Fig. 2. We first delete 4 from the tree. (We were lucky that it was a leaf. If not, we would need to first find the replacement node, copy its contents, and delete the replacement node.)

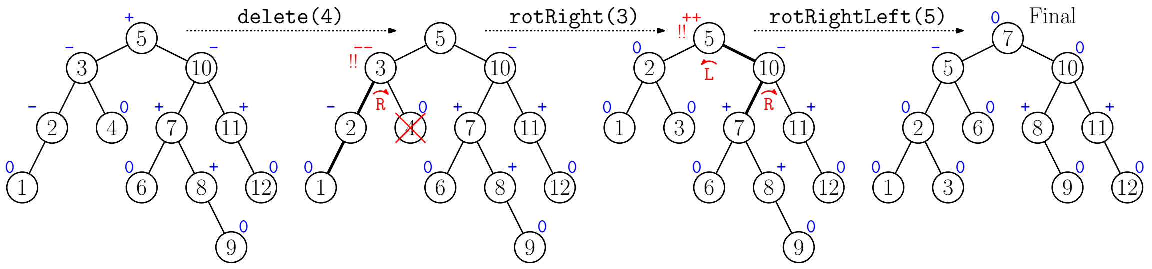

Figure 2: AVL tree: `delete(4)`.

We then retrace the search path back to the root, updating balance factors as we go. This decreases the balance factor at 3 to -2, which is not valid. We find that 3 is left-left heavy, that is, its height is determined by its left-left grandchild, 1. (Remember when there are ties for heaviness, we favor single rotations over double rotations.) This induces a right single rotation at 3, which brings the 2 up. We update the balance factors of the affected nodes (2 and 3). We then continue retracing.

Next we increase the balance factor at 5 to +2, which is invalid. We find that it is right-left heavy, that is, its height is determined by its right-left grandchild, 7. This induces a right-left rotation at 5, which performs a right rotation at 10 and a left rotation at 5. We then update the balance factors of the affected nodes (5, 7, and 10). Node 7 is the new root of the tree, and we are done.

Solution 2:

(a) See Fig. 3. The insertion adds a red left child to 23 at level 1. When we perform `skew(23)` we discover this and do a right rotation at 23. Next, when we perform `split(21)` we find that its right-right grandchild (25) is at the same level (red), and so we do a left rotation at 21 and promote its right child (23) to level 2. We return to 27 and perform `skew(27)`. It sees that its left child (23) is red, and so we do a right rotation at 27. We return to 20, and when we perform `split(20)`, we discover that its right-right grandchild (27) is at the same level (red), and so we do a left rotation at 20 and promote its right child (23) to level 3. We return to 15 (which is fine) and finally to 6. When we perform `split(6)`, we see that its right-right grandchild (23) is at the same level, and so we do a left rotation at 6 and promote its right child (15) to level 4. Node 15 becomes the new root.

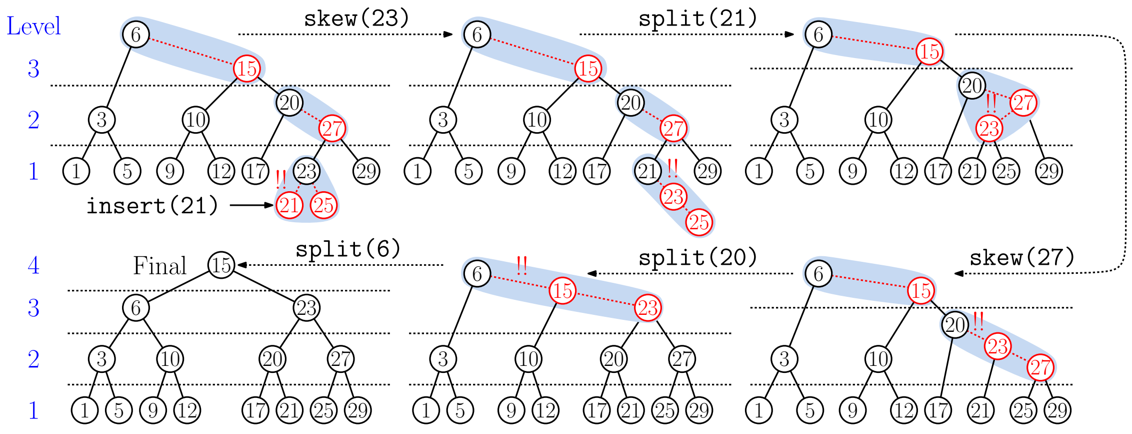

Figure 3: AA Tree: `insert(21)`.

(b) See Fig. 4. After deleting 13, we perform `update-level` on 14. Its left child (`nil`) is at level 0 and so it is pulled down to level 1. Its right child (18) comes down with it. Now we have a "train wreck" of five nodes at level 1. We perform the prescribed series of skews and splits. The second skew, `skew(18)`, discovers that it has a red left child (15), and so it does a right rotation, which brings this child up. Now the train wreck is nicely lined up on the right. Next, we perform `split(14)`, where we discover that its right-right grandchild (18) is at the same level (red), and so we do a left rotation at 14 and promote its right child (15) to level

2. Node 15 becomes the current node, and when we perform `split(p.right) = split(18)` we discover that its right-right grandchild (20) is at the same level (red), and so we do a left rotation at 18 and promote its right child (19) to level 2, where it becomes the right child of 15. We return up through the tree 15, then 11, then 4, but none of these nodes need to be changed.

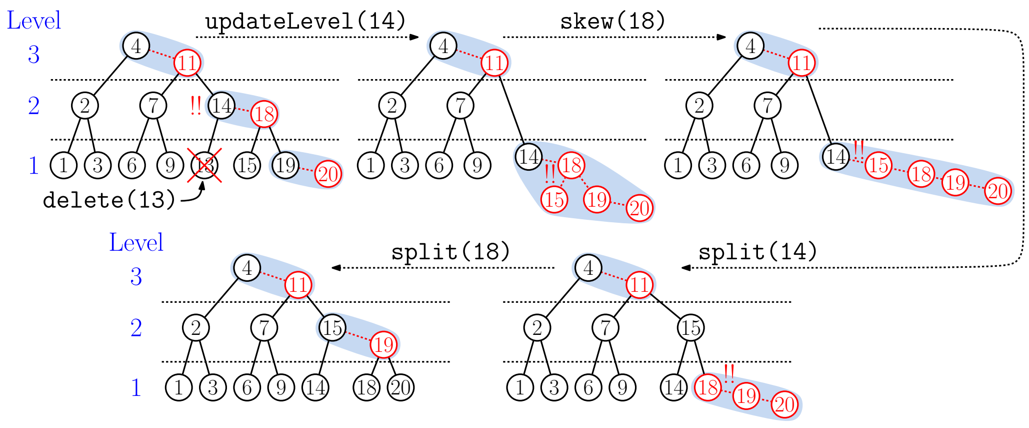

Figure 4: AA Tree: `delete(13)`.

Solution 3:

(a) The 2-3-4 tree is shown in the upper left of Fig. 5.

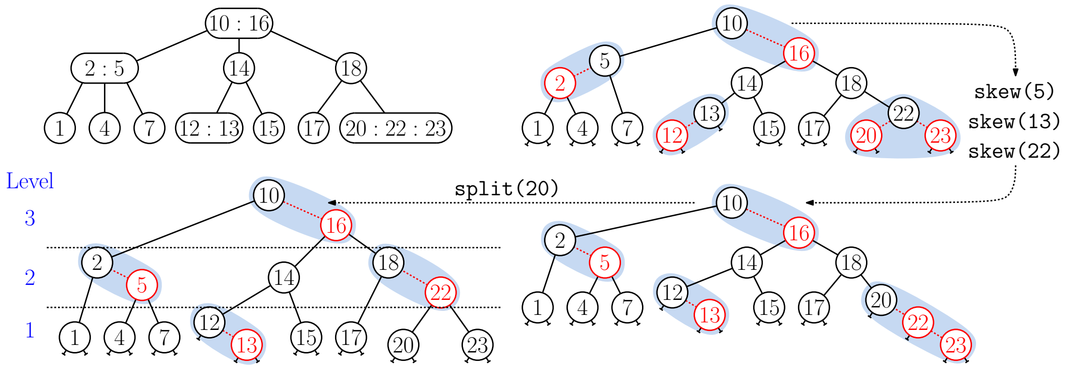

Figure 5: Converting a red-black tree to a 2-3-4 tree and an AA tree.

(b) To convert the red-black tree to an AA tree, we first identify all instances where we have a red left child and we perform skews at all of these nodes (`skew(5), skew(13), skew(22)`). Following this, we check to see if we have any right-right chain of red nodes, and we remedy these by performing a split at the topmost node (`split(20)`). In general, the latter operation might generate an instance where further skewing or splitting is necessary, but in this case, the tree is a valid AA tree, and so we are done.

Solution 4:

(a) To perform `insert(9)`, we start with `splay(9)` (see Fig. 6). This begins by searching for 9 in the tree. We descend until falling out of the tree at 10's left child. We start the splaying process at 10. It is a right-right grandchild of 4, so we perform zig-zig rotation involving the nodes 10, 8, and 4. This pulls the 10 up to the top and makes 8 its child and 4 is grandchild. Now, 10 is a right-left grandchild of 3, so we perform a zig-zag rotation involving 10, 12 and 3. This moves the 10 up and makes 3 and 12 its children. Finally, 10 is the left child of the root, so we perform a final zig operation. This makes 10 the root of the tree and 17 becomes its right child.

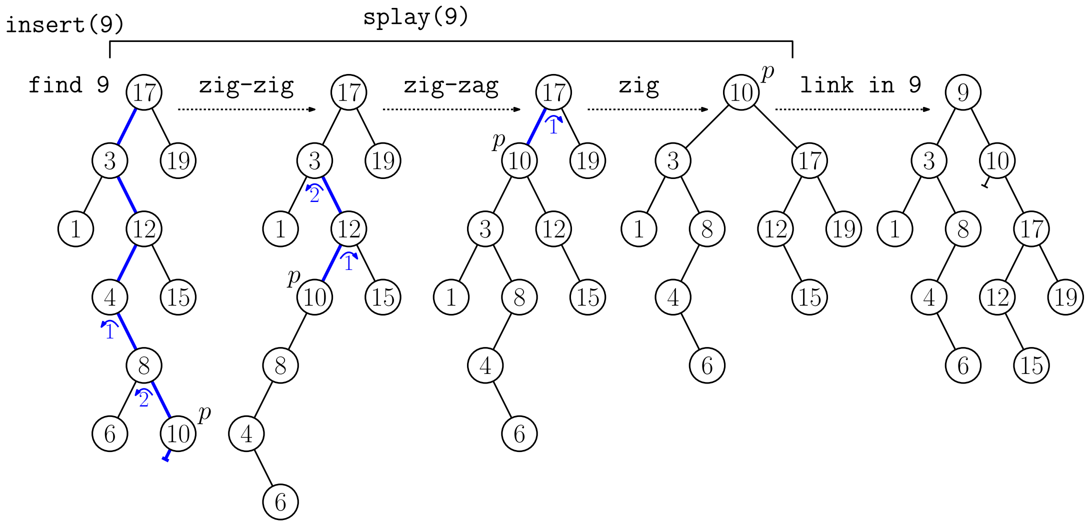

Figure 6: Splay Tree: `insert(9)`.

In this case, 10 is the successor of 9. So, to link 9 into the tree we create a root node containing 9 and make 10 its right child. (10's left child is set to `null`.)

(b) To perform `delete(8)`, we start with `splay(8)` (see Fig. 7). This searches for 8 in the tree. Since 8 is a left-right grandchild of 10, we perform a zig-zag rotation involving 8, 6, and 10. This brings up the 8 and makes 6 and 10 its children. Now, 8 is the left-left grandchild of 17, so we perform a zig-zig rotation, which brings up the 8 and makes 12 its right child and 17 its right-right grandchild. Now 8 is the right child of the root 3, and so we perform a zig rotation, which brings 8 up to the root and makes 3 is left child.

Now that 8 is at the root, we need to find the replacement. We perform `splay(8)` on its right subtree. (You could have done this on the left subtree, but this is the convention we follow.) The search for 8 falls out of the tree on the left child of 10. 10 is the left child of the subtree root 12, so we perform a zig rotation which makes 10 the new right child of 8. Note that 10's left child is `null` (which must be true, since 10 is the smallest key in the right subtree). We unlink 8 by making 10 the new root.

(c) (This was not required, but I've just included it.) To perform `insert(5)`, we start with `splay(5)` (see Fig. 8).

This begins by searching for 5 in the tree. We descend until falling out of the tree at 6's left

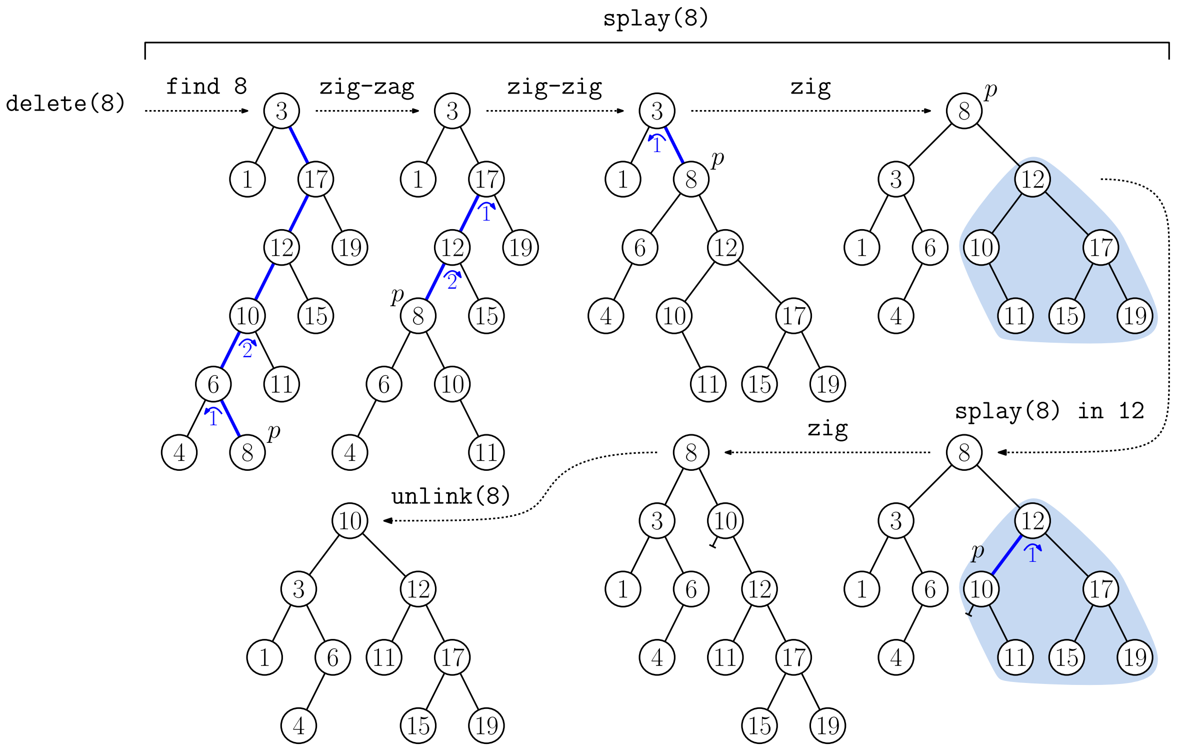

Figure 7: Splay Tree: `delete(8)`.

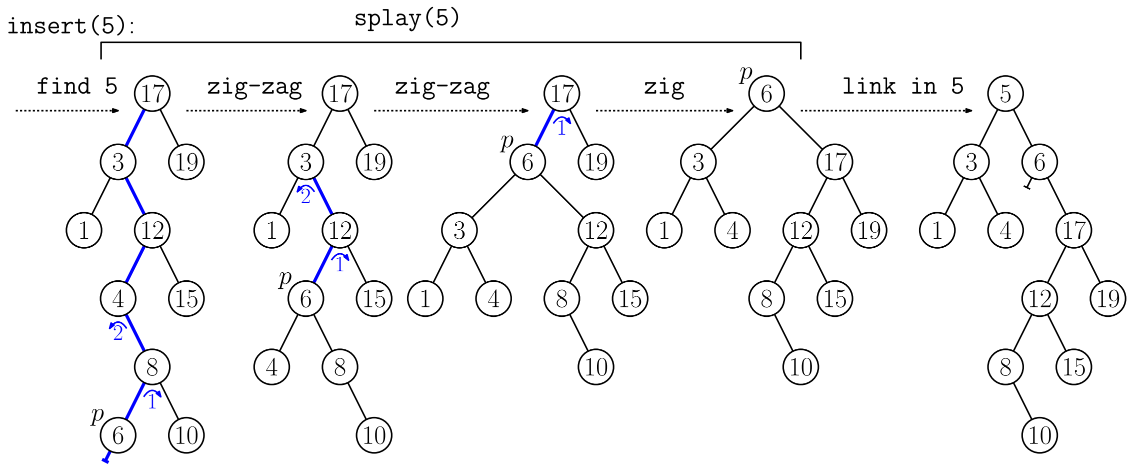

Figure 8: Splay Tree: `insert(5)`.

child. We start the splaying process at 6. It is a right-left grandchild of 4, so we perform zig-zag rotation involving the nodes 6, 8, and 4. This moves the 6 up and makes 4 and 8 its children. Now, 6 is a right-left grandchild of 3, so we perform a zig-zag rotation involving 3, 12, and 6. This moves the 6 up and makes 3 and 12 its children. Finally, 6 is the left child of the root, so we perform a final zig operation. This makes 6 the root of the tree and 17 becomes its right child.

In this case, 6 is the successor of 5. So, to link 5 into the tree we create a root node containing 5 and make 6 its right child. (6's left child is set to `null`.)

Solution 5:

(a) This is quite an interesting proof. I will present two versions, both by induction. The first employs a bottom-up approach by removing nodes from the bottom of the tree. The second employs our usual top-down approach by joining two subtrees under a common root node.

In either case, our objective is to show that, given the preorder traversal of any full binary tree with $n \geq 1$ nodes and a labeling indicating which nodes are leaves and which are internal, there is a unique full binary tree having this traversal.

The basis case is when $n = 1$, that is, we have just a single leaf node. Since there is only one tree on a single node, the basis case holds trivially. Otherwise, let us make the (strong) induction hypothesis that $n > 1$, and for any n', where $1 \leq n' < n$, a preorder traversal and leaf labeling uniquely determines the structure of a full binary tree of size n'. Our goal is to use this to prove the result for n itself.

Bottom-Up Proof: Consider a traversal consisting of n nodes along with the leaf labels. We first claim that for any full binary tree having at least one internal node, the preorder list must contain the consecutive leaf-label triple $\langle F, T, T\rangle$. (For example, in Fig. 9 this occurs for the triples $\langle d, h, i\rangle$, $\langle f, j, k\rangle$ and $\langle m, n, o\rangle$.) We also claim that whenever this pattern occurs, it represents a three-node subtree. (For example, it cannot be the case that the first F and T belong to one subtree and the last T belongs to a completely different subtree.)

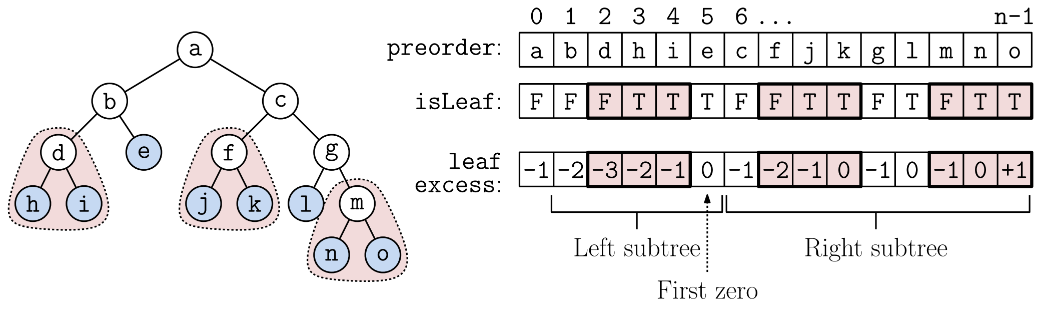

Figure 9: Uniquely reconstructing a tree from its preorder traversal.

To see why this pattern must occur at least once, consider the deepest internal node in the tree. By fullness, its two children must be leaves. So, when the preorder traversal arrives at this node, it visits the internal node (F), then its left-child leaf (T), then its right-child leaf (T), yielding the triple $\langle F, T, T\rangle$. Next, to see that such a triple must form a subtree, consider what happens when the traversal arrives at the first node (F). We know that it is internal. By fullness, we know that it has both children. Because the next label is T, the left child is a leaf, and because the next label to that is T, we know the right child is a leaf. Therefore, this forms a three-node subtree.

Using these two claims, the proof proceeds as follows. First, find any instance of $\langle F, T, T\rangle$ among the leaf labels. We know that in *any* reconstructed tree, this sequence must form a three-node subtree. Remove the triple (from both the traversal and the leaf labeling), and replace it with a single leaf node called x. Observe that the resulting tree is full. Also, because we have replaced three nodes with one, the resulting traversal has $n' = n - 3 + 1 = n - 2$ nodes. Since we have strictly fewer than n nodes, we can apply the induction hypothesis, which tells us that there is a unique full tree corresponding to this new traversal. Construct this tree. Now, locate the leaf node x in this tree, and restore the original three-node subtree. The resulting tree is uniquely determined from the traversal.

Top-Down Proof: Next, let's consider how to prove this in a top-down manner. Suppose that we are given a preorder traversal `preorder[0..n-1]` for a tree with $n > 1$ nodes. The first element of the traversal is clearly the root. The remainder of the array will then be split into two subarrays, the first, `preorder[1..j-1]`, contains a preorder traversal of the left subtree, and the second, `preorder[j..n-1]`, contains preorder traversal of the right subtree. (For example, in Fig. 9 we have $j = 6$.) How do we know that there is a unique place where this split can made? Let's just assume this for now, and fix this issue up later.

Assuming that the index j where the array is split is uniquely determined, we can now complete the induction proof. Each of `preorder[1..j-1]` and `preorder[j..n-1]` are of sizes strictly smaller than n, and so the induction hypothesis applies to them. That is, each yields a unique subtree, and putting them together with the first element (the root) yields the unique final tree—QED.

All that remains is to show the uniqueness of the splitting point, we start with a useful fact about full binary trees, namely that if a full tree has one more leaf than it has internal nodes. (We proved this for extended binary trees in Lecture 3, but a full binary tree is essentially the same as a extended binary tree, where the leaves are the extensions of the internal nodes.)

While we are performing a preorder traversal, define the *leaf excess* to be the number of leaves seen in the traversal so far minus the number of internal nodes (See Fig. 9). By the above observation, when the traversal of a full tree is finished, the leaf excess is exactly $+1$. We assert that during a preorder traversal of a full tree, the first time the leaf excess is exactly 0 comes just after visiting all the nodes in the left subtree. This condition uniquely identifies the node `preorder[j-1]` where we transition from the left subtree to the right subtree.

To see this, first observe that when we encounter the last leaf node of the left subtree, this entire subtree contributes a net of $+1$ to the leaf excess, and the root node contributes -1, so the total excess is zero. To see that the excess cannot be zero prior to this, observe that we start with -1 as soon as we visit the root. After this, every subtree that we have fully visited contributes $+1$ to the count, but the parent of this subtree contributes -1 for a total of zero. Thus, we stay in negative territory. It is only when we complete the root's entire left subtree, that we get back to zero. This condition uniquely identifies the index j separating the left and right subtrees.

(b) We present pseudocode for a recursive helper function called `buildTree`. It is given the index i of the current node, and we return a pointer to the subtree rooted at this node and advance i to the next node in the traversal following this subtree. Note that i is a reference parameter, so modifying i in the function modifies its value. The initial call is `buildTree(0)`. We assume that `preorder` and `isLeaf` are global variables.

If the node is a leaf, we simply create a new leaf node, which we return. Otherwise, we consume the first entry, which becomes the root of the subtree and recursively invoke the procedure to build the left subtree and then to build the right subtree.

```
Node buildTree(ref int i) {  // note that i is a reference parameter
    if (isLeaf[i]) {
        Node u = new Node(preorder[i], null, null)  // create a new leaf node
        i++; return u
    } else {
        Node u = new Node(preorder[i], null, null)  // create a new internal node
        i++
        u.left = buildTree(i)                       // build its left subtree
        u.right = buildTree(i)                      // build its right subtree
        return u
    }
}
```

The correctness follows from (a) and the assumption that the inputs represent a valid preorder traversal of a full tree.

Solution to the Challenge Problem: Given a structurally valid AA without its level numbers, can the level numbers be uniquely reconstructed? The answer is yes, and the way in which this is done is remarkably simple.

Traverse the nodes in postorder. This implies that whenever we visit a node `u`, its two children have already been visited, and we may assume that their levels have been computed. Define a function `level(u)` which returns `u.level` if `u` is non-null and zero otherwise. (Alternatively, we could have employed the `nil` sentinel node as used in AA trees.) We can then compute `u`'s level number as

$$\texttt{u.level} \;=\; 1+\min(\texttt{level(u.left)},\texttt{level(u.right)}).$$

(It is interesting to note that this function is essentially the same as the NPL function used in Leftist Heaps, except that we start counting one level higher.)

Why is this correct? First off, notice that this is the largest level number that the `updateLevel` function will allow a node to have. So (assuming the tree is a valid AA tree), the level number certainly cannot be larger than this. However, it cannot be smaller than this either, for if it were, then it would have the same or smaller level number as both its children. A node certainly cannot have a level number smaller than any child. If a node's level number is the same as both children, then this node has two red children. But this is not allowed in any valid AA tree. Since the level number cannot be larger, and it cannot be smaller, this formula yields the correct level number.