CMPT 280

Tutorial: Efficient AVL Trees

Mark G. Eramian

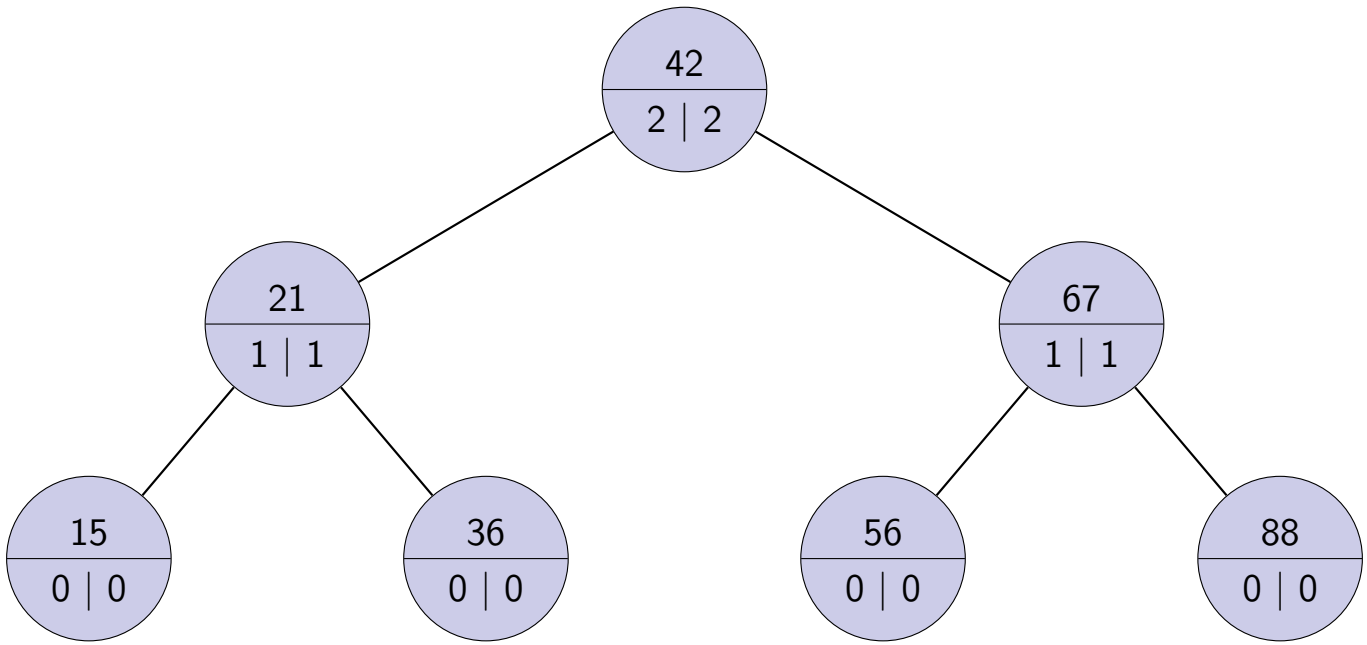
University of Saskatchewan

Efficient Insertion and Deletion from AVL Trees

- As discussed in class, in order to achieve $\Theta(\log n)$ insertion and deletion algorithms for AVL Trees, we need to be able to look up the height of a node's left and right subtrees in constant time so that we can efficiently test for critical nodes after an insertion or deletion.
- Solution: Store in each node the heights of it's subtrees.
- In these slides we will look at how this is done, conceptually, and how these stored heights are updated after an insertion or deletion.
- We'll also do some more example insertions and deletions with rotations.

Efficient Insertion for AVL Trees

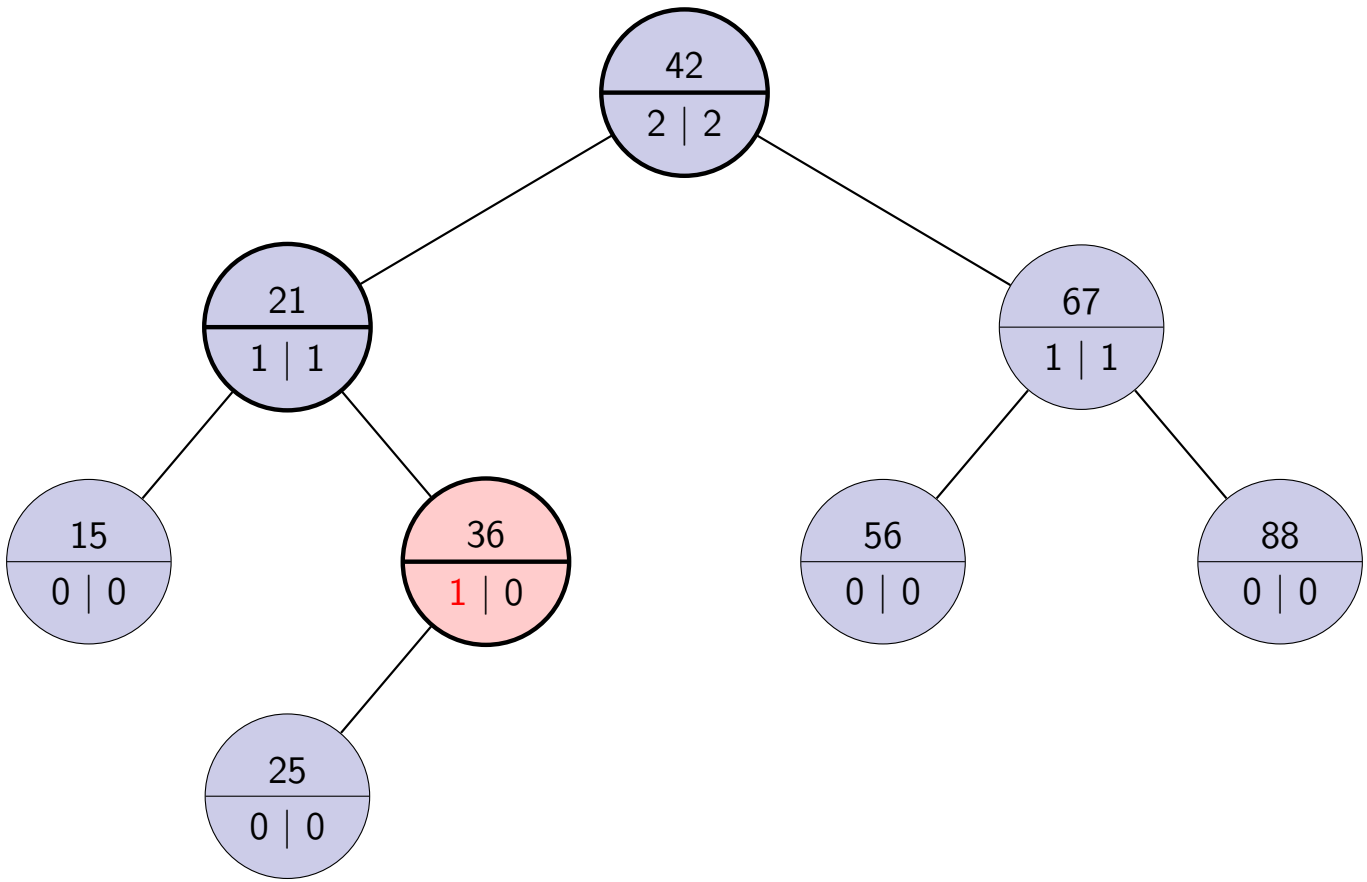
Let's insert 25 into this tree. The lower half of each node show its left and right subtree heights.



Efficient Insertion for AVL Trees

Inserting 25

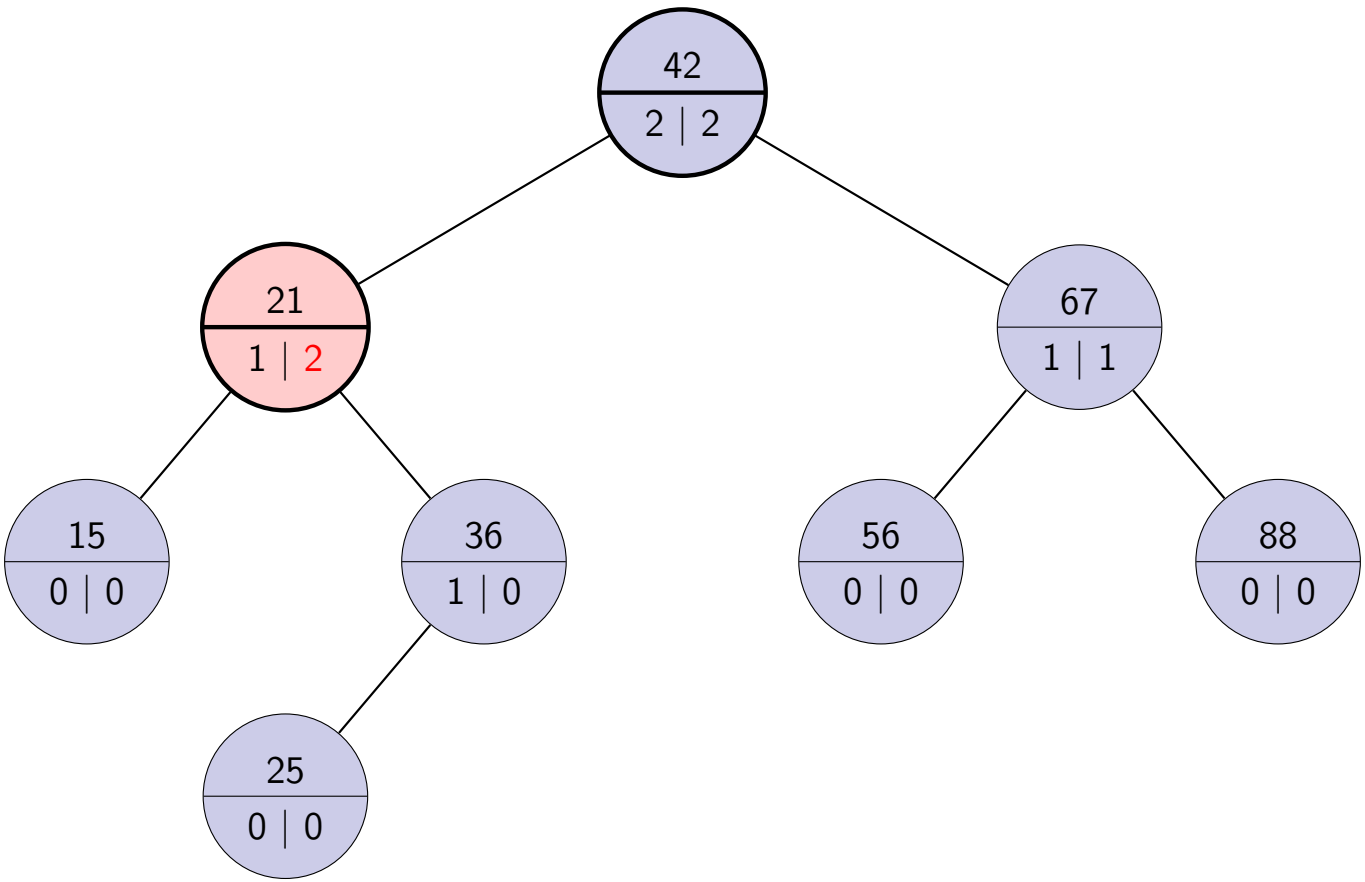
Recurse to 36, insert 25 as left child as part of the base case of the recursion. We know 36's left subtree height must now be 1. Check for critical node (nope, node 36 has an imbalance of 1).



Efficient Insertion for AVL Trees

Inserting 25

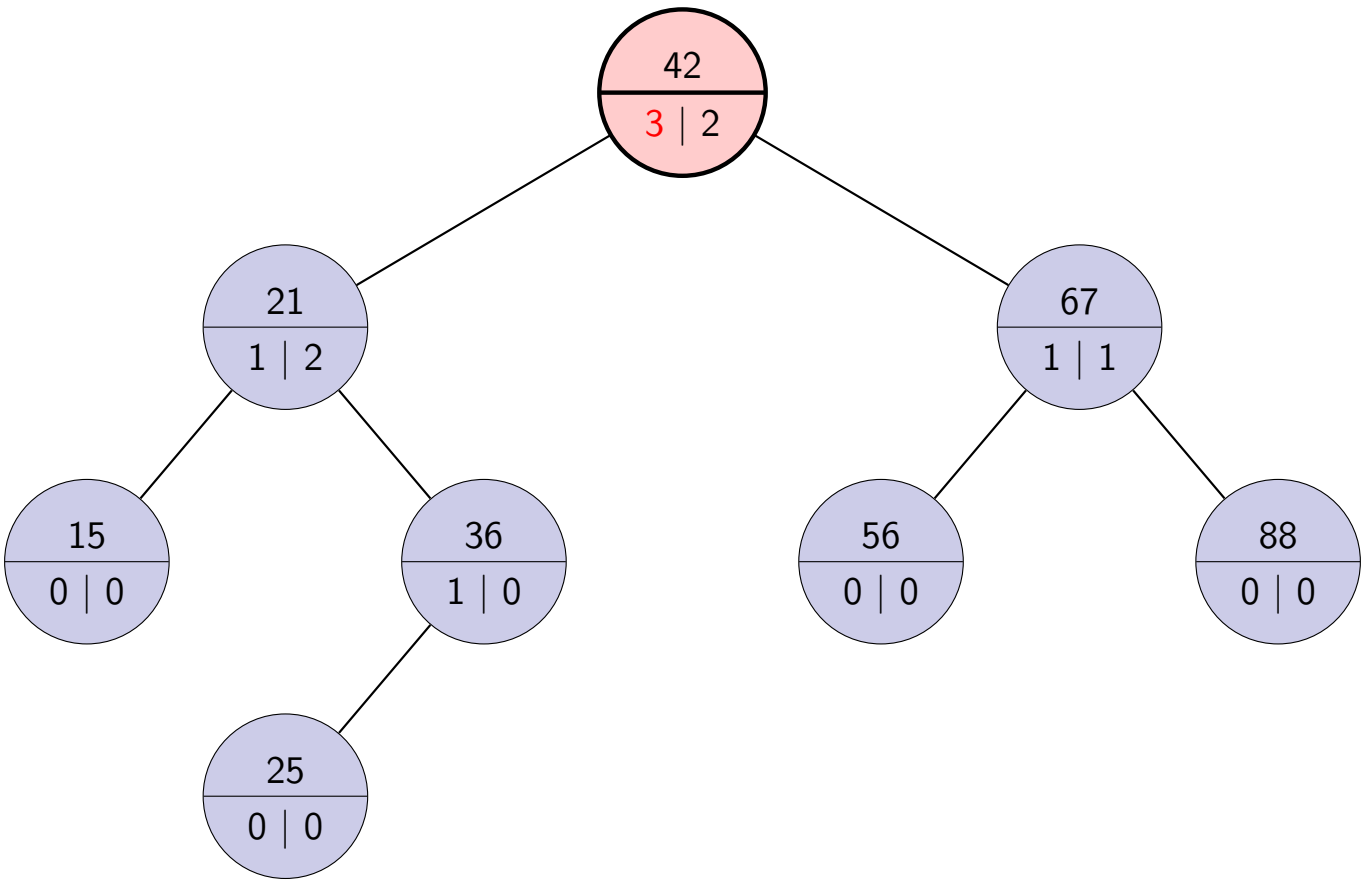
Return to 21. Recompute height of the subtree we returned from as: max(rightchild.leftSubtreeHeight + rightchild.rightSubtreeHeight) + 1, check for critical node (it's not: new imbalance at node 21 is 1).



Efficient Insertion for AVL Trees

Inserting 25

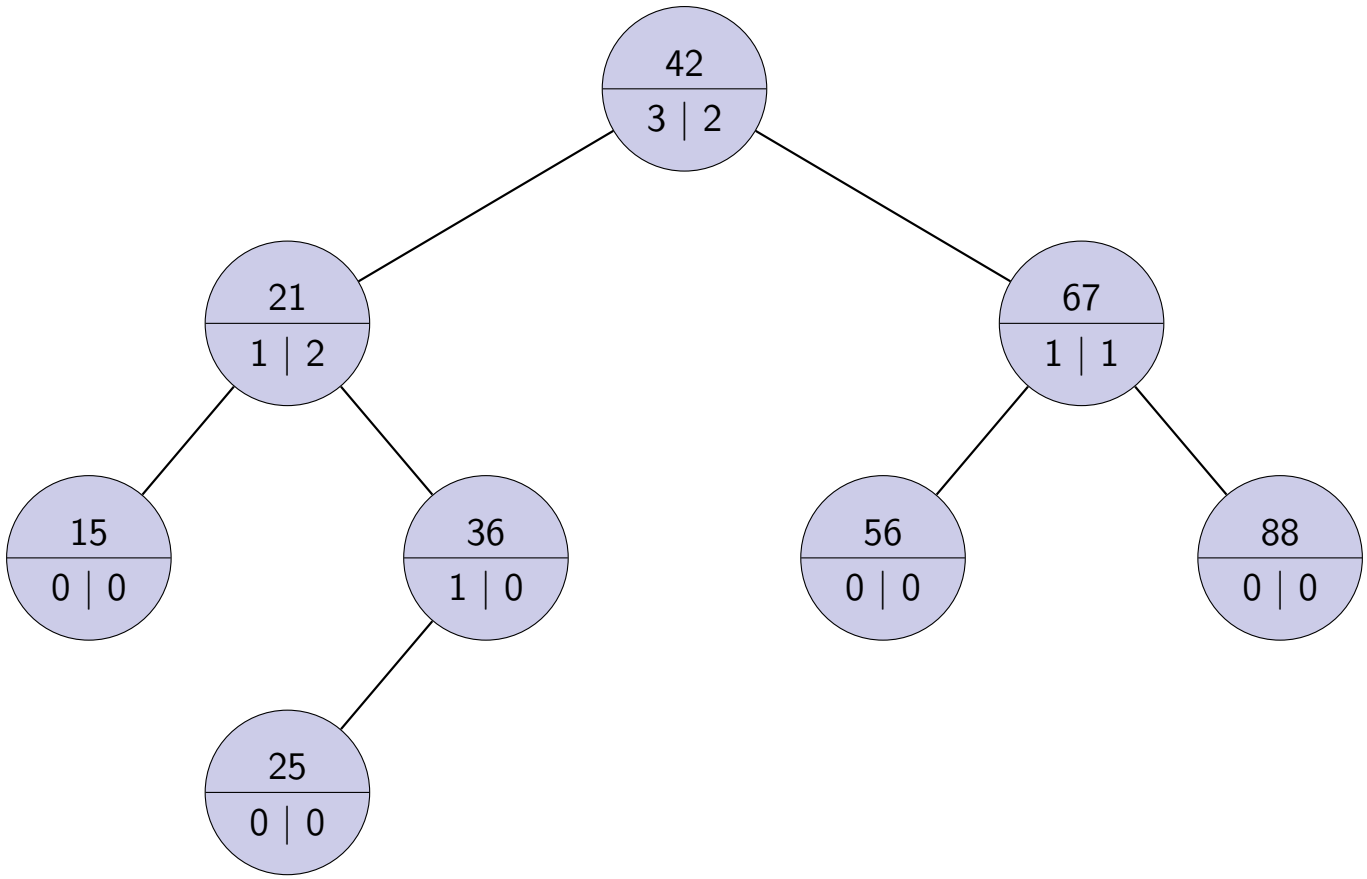
Return to 42. Recompute height of the subtree we returned from as: max(leftchild.leftSubtreeHeight + leftchild.rightSubtreeHeight) + 1, check for critical node (it's not: new imbalance at node 42 is 1).



Efficient Insertion for AVL Trees

Inserting 31

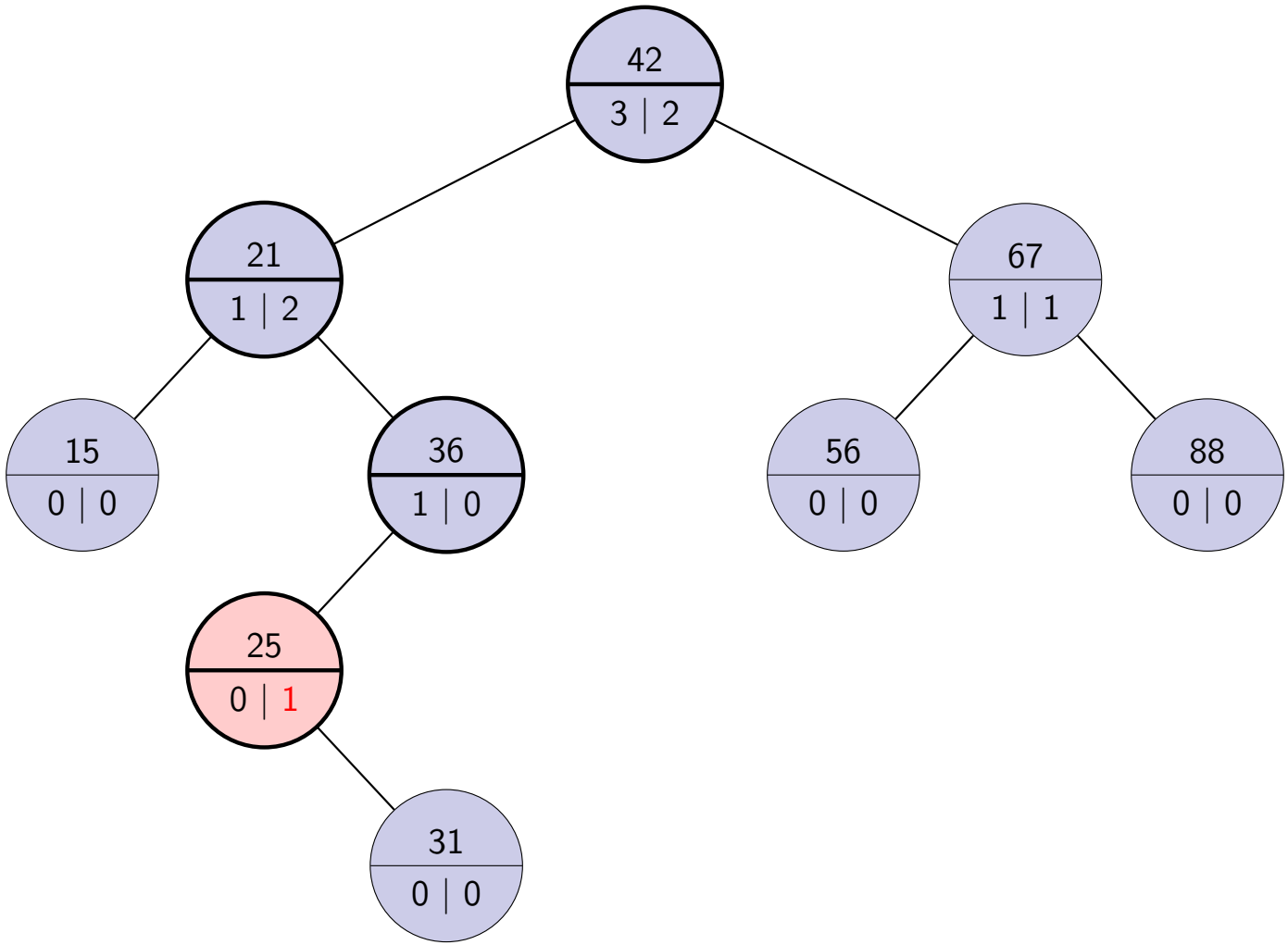
Now insert 31.



Efficient Insertion for AVL Trees

Inserting 31

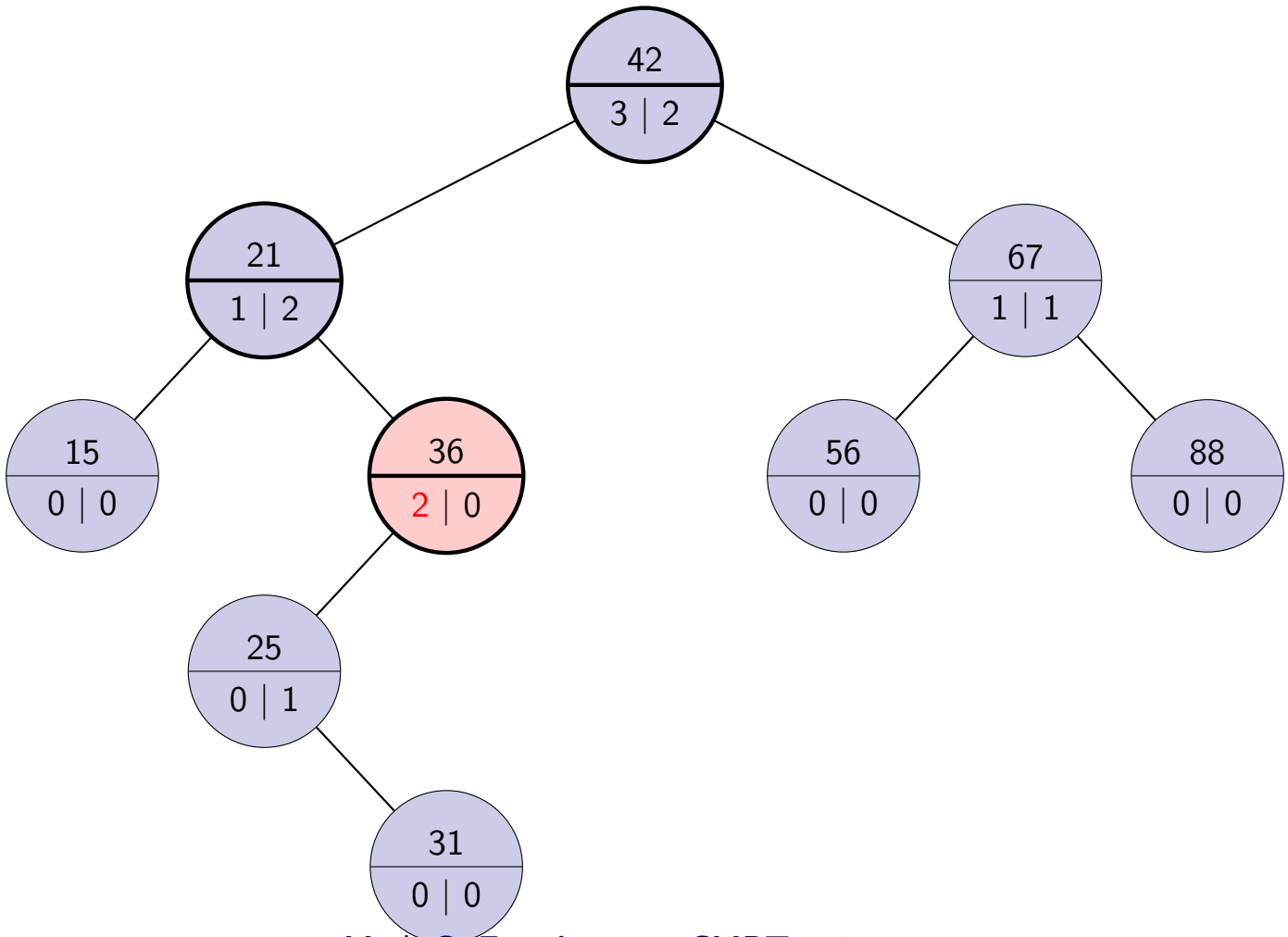
Recurse to node 25 (base case). Insert 31 as right child of 25. Record known height of 1 for right subtree of 25. Check for critical node (nope, 25 has an imbalance of 1).



Efficient Insertion for AVL Trees

Inserting 31

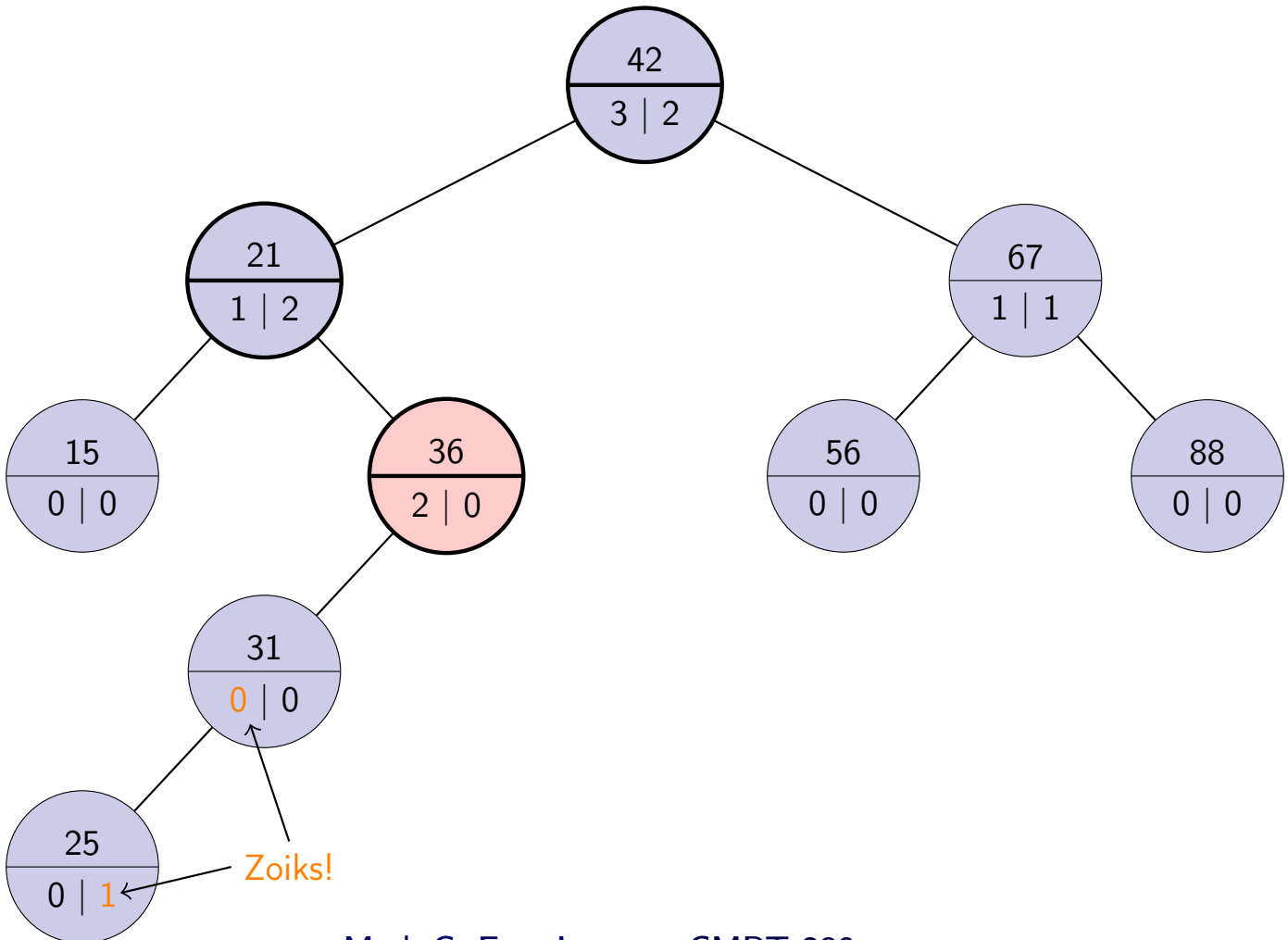
Return to 36. Recompute height of left subtree: max(leftChild.leftSubtreeHeight, leftChild.rightSubtreeHeight) + 1 = 2. Check for critical node: YES, 36 now has an imbalance of 2 and exhibits LR imbalance. We need to perform a double right rotation.



Efficient Insertion for AVL Trees

Inserting 31

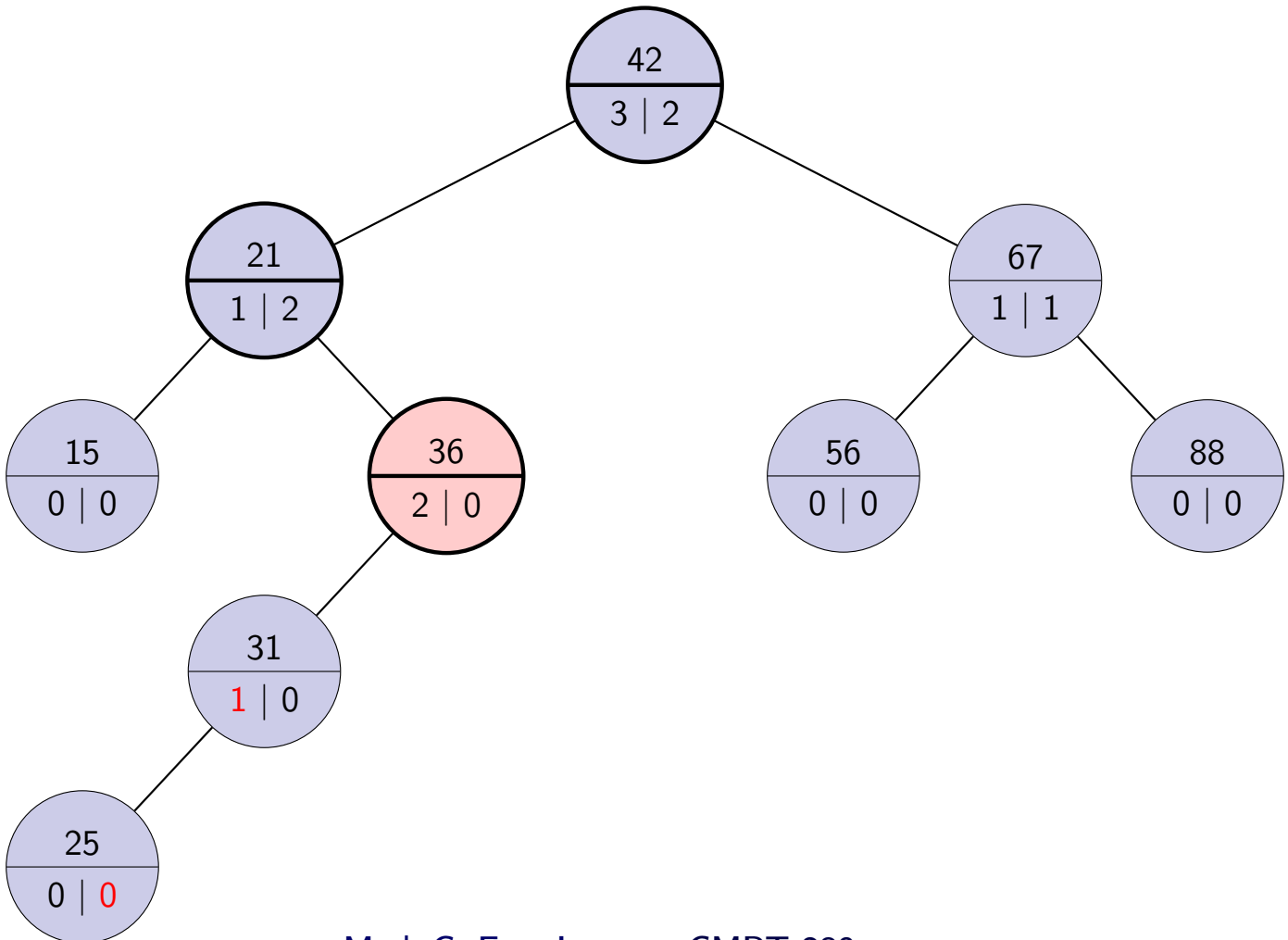
Double right rotation at 36 is required. First step, left rotation of 25. Uh oh, now tree heights are wrong again! After a rotation we need to recompute the subtree heights for any node whose subtrees changed as the result of the rotation. This occurs as before...



Efficient Insertion for AVL Trees

Inserting 31

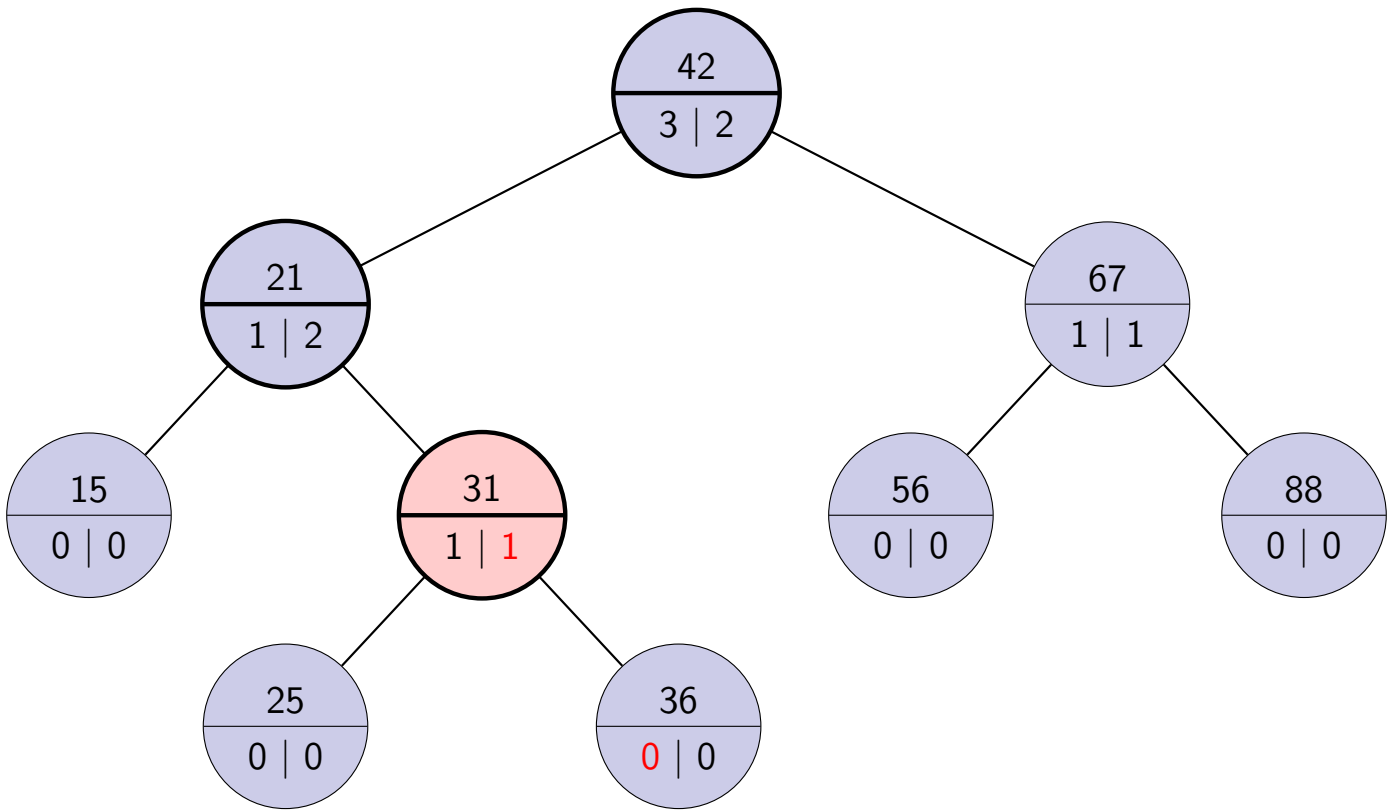
25's right subtree height is 0 because it's left subtree reference is null. 31's left subtree height is $\max(0, 0) + 1 = 1$. (Larger of node 25's two subtree heights plus 1). Now we can finish the double right rotation by doing a right rotation of 36...



Efficient Insertion for AVL Trees

Inserting 31

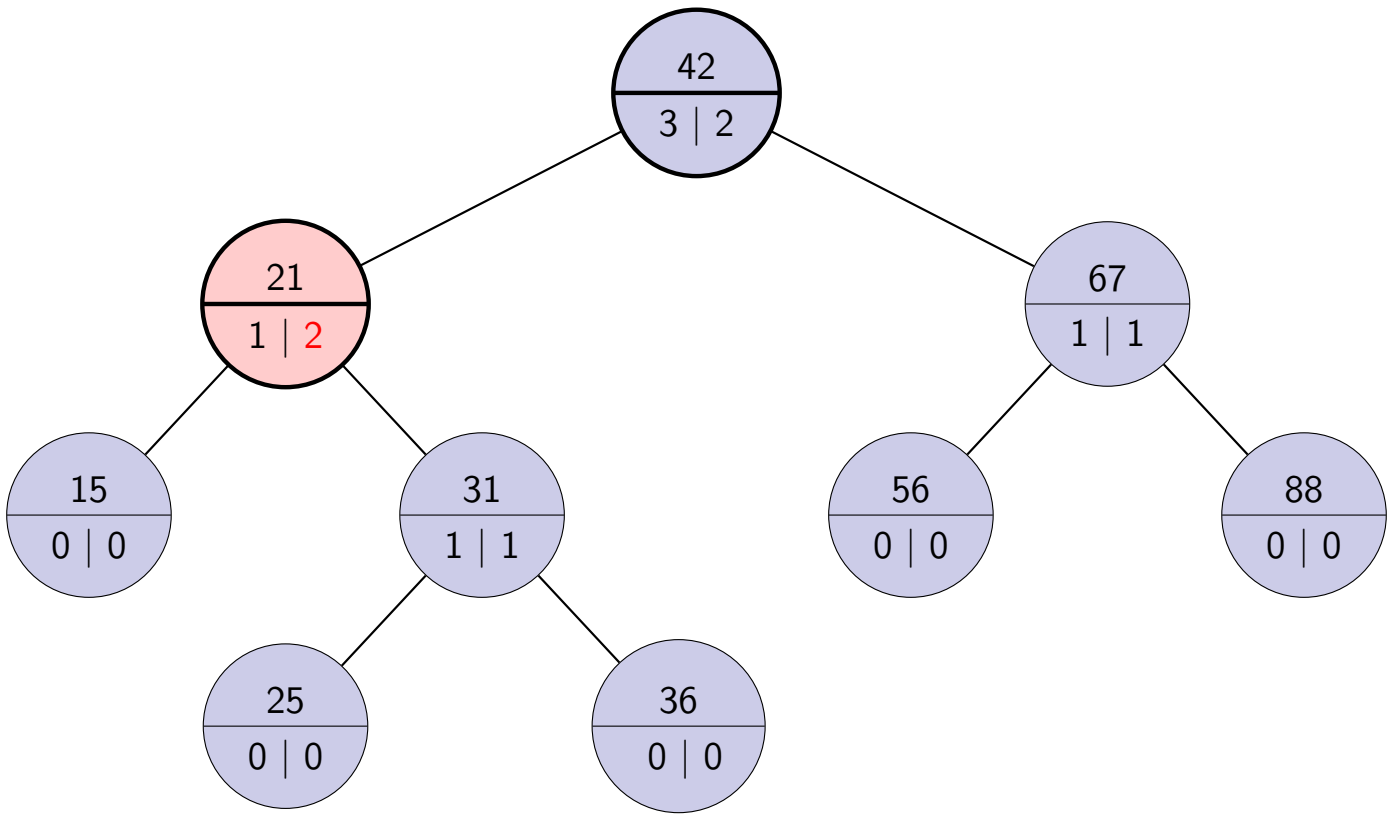
A right rotation of 36 completes the double right rotation. But again, subtree heights for any subtrees that changed as a result of the computation need to be recomputed from the heights stored in the roots of those subtrees (there will always be two subtree heights that need updating for any single rotation!).



Efficient Insertion for AVL Trees

Inserting 31

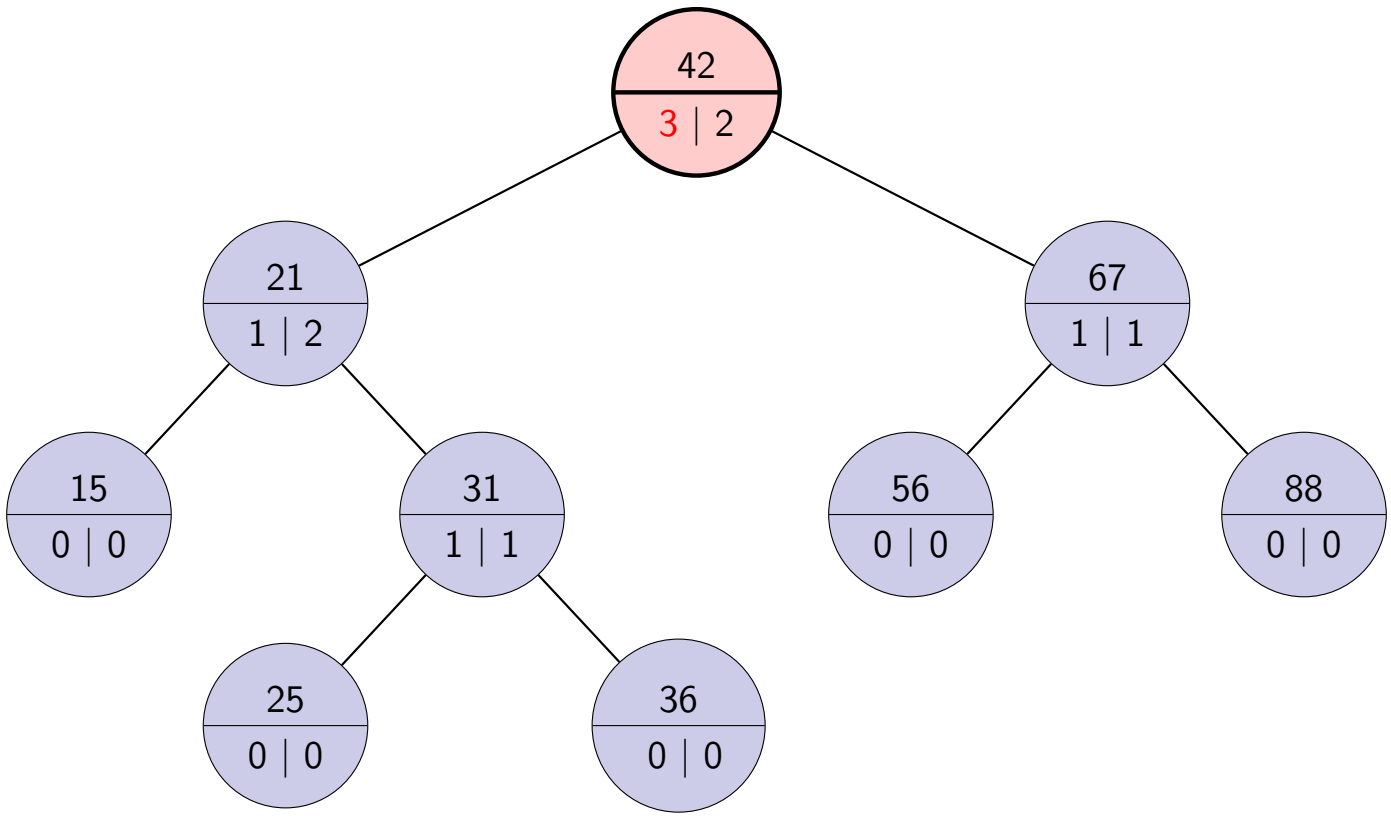
Now we return to 21 and recompute 21's right subtree height from the heights stored in the root of that subtree (which is 31). The new height of 21's right subtree is $\max(1, 1) + 1 = 2$, which happens to be the same as before. We check for a critical node and find that node 21's imbalance is 1, so no rotations are necessary.



Efficient Insertion for AVL Trees

Inserting 31

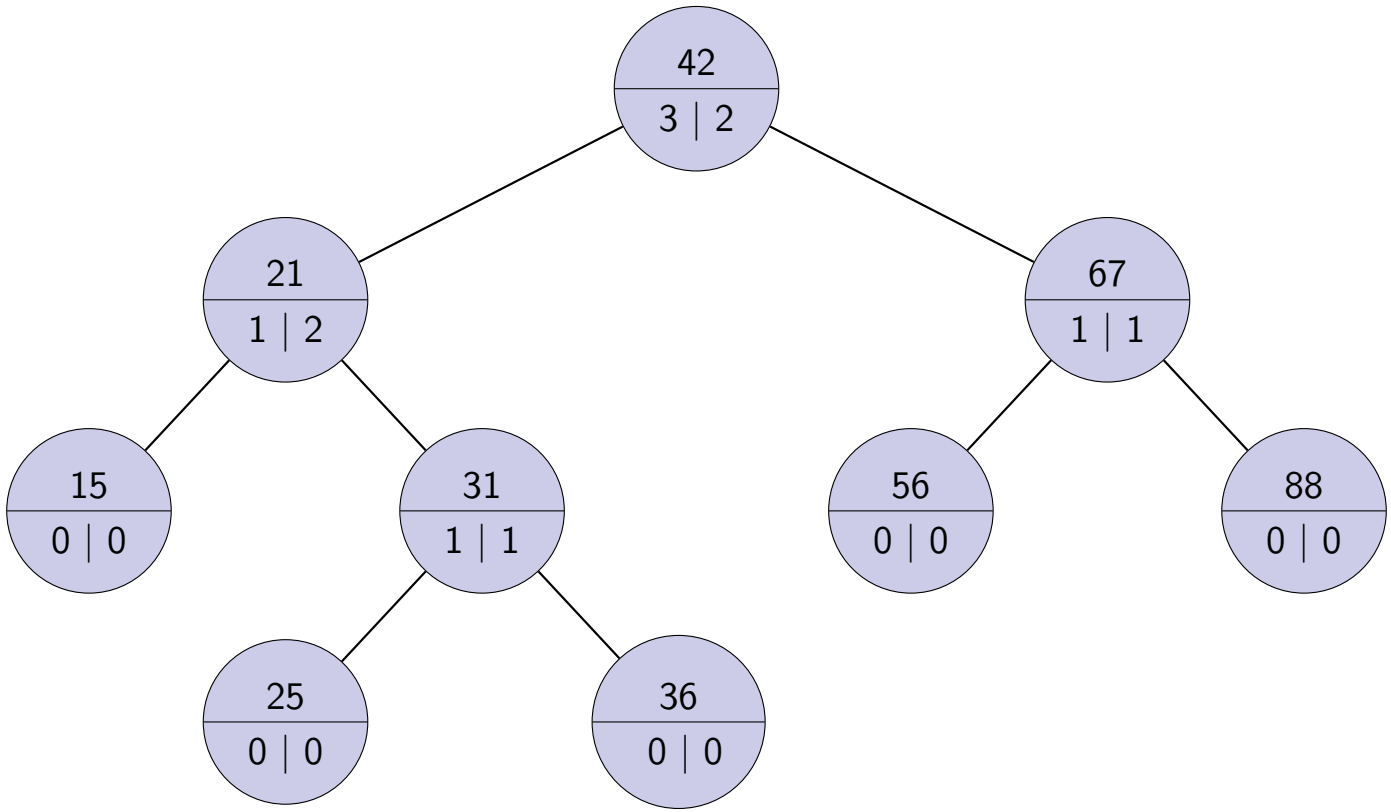
Now we return to 42. Since we returned from the left subtree we must recompute the height of 42's left subtree from the root of that subtree, which is 21. Thus, the new height of 42's left subtree is $\max(1, 2) + 1 = 3$, which, again, happens to be the same as before. We have to recompute it though, because there are situations where it might **not** be the same.



Efficient Insertion for AVL Trees

Inserting 31

We return again, and the insertion of 31 is complete. All rotations were performed, and all subtree heights were kept up to date.



Efficient Insertion for AVL Trees

- Efficient insertion therefore is exactly as the algorithm discussed in class with the addition of:
    - Each time you return from a recursive call to insert(), recompute the height of the subtree you just returned from before calling restoreAVLProperty() to check for a critical node and perform rotations.
    - Each time a rotation occurs, recompute the subtree height for any reference that was modified as a result of the rotation.

Efficient Insertion for AVL Trees

- Remember, to recompute the height of the left subtree of a node v, take the larger of the two heights from the left child of v and add one.
- Likewise to recompute the height of the right subtree of a node v, take the larger of the two heights from the right child of v and add 1.
- In either case above, if the child of v does not exist, then the height of the subtree in question is, of course, just 0.

Efficient Deletion for AVL Trees

- Everything we discussed about maintaining subtree heights during insertion applies to deletion as well.
- When a recursive call to delete() returns, the height of the subtree from which we just returned must be recomputed before checking whether the current node is critical.
- Subtree heights must be recomputed for any subtrees that change during rotations (this is already done if you already have rotations coded correctly for insertion).

More Examples

- We can do some more examples on the board...