More Examples of Operations on AVL Tree

Tony Gong

ITEE
University of Queensland

Recall in lectures we studied the AVL tree, which is one type of self-balancing binary search tree. The aim was to store a set of integers S supporting the following operations:

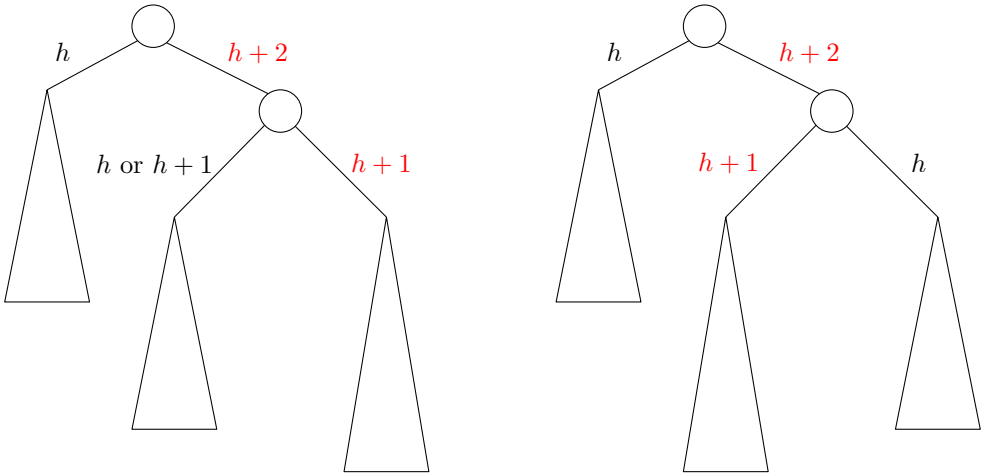
- A predecessor query: given an integer q, find its predecessor in S;
- Insertion: add a new integer to S; and
- Deletion: remove an integer from S.

We want all of these operations to run in $O(\log n)$ (where n is the number of integers in S) in the worst case. If we were to attempt to accomplish this using a BST, we must ensure it is balanced after every operation, and the AVL tree presents one method of doing so.

Rebalancing

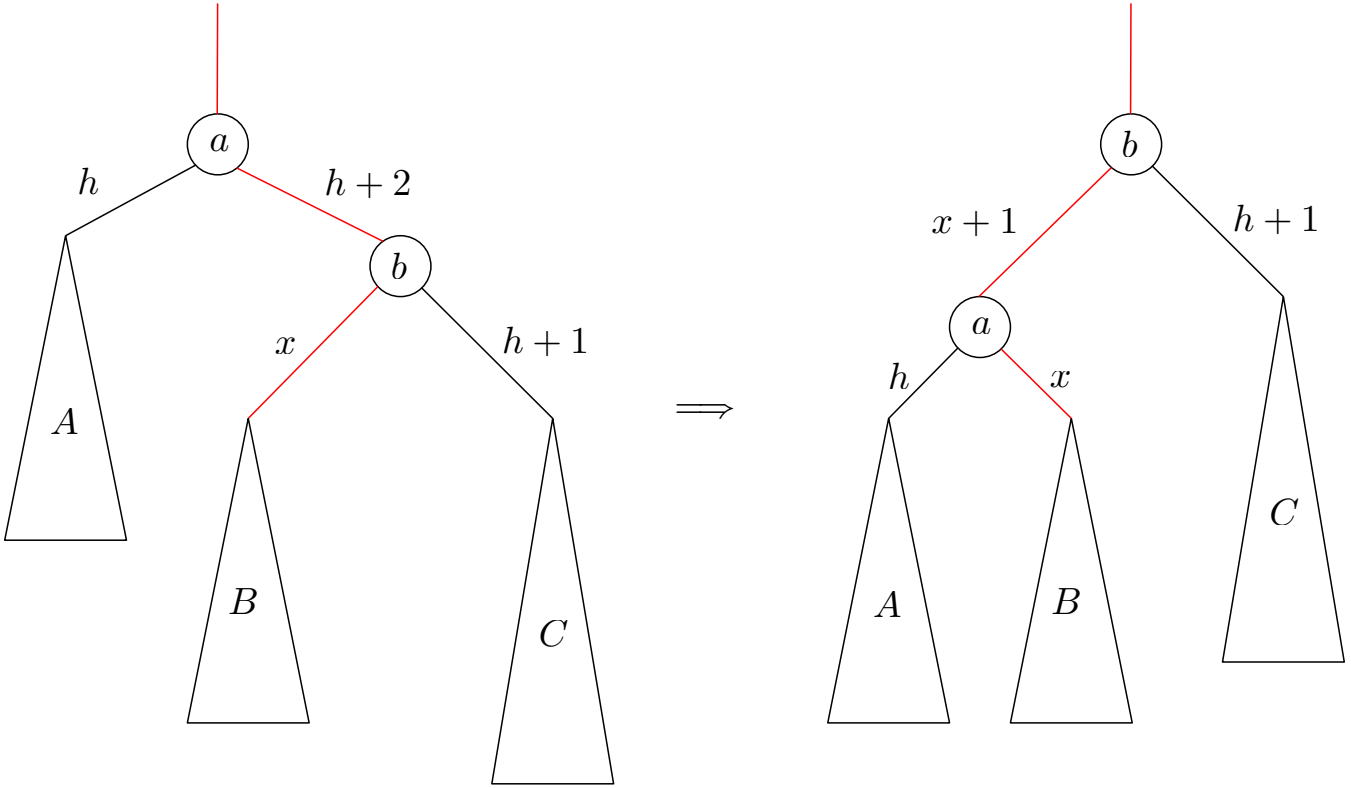
We know that a tree is balanced as long as the height of its subtrees differ by at most 1, and that insertion and deletion can only cause a 2-level imbanace (where the heights differ by 2).

In lectures we explored the Left-Left and Left-Right cases in detail, so here we will look at Right-Right and Right-Left:



Right-Right

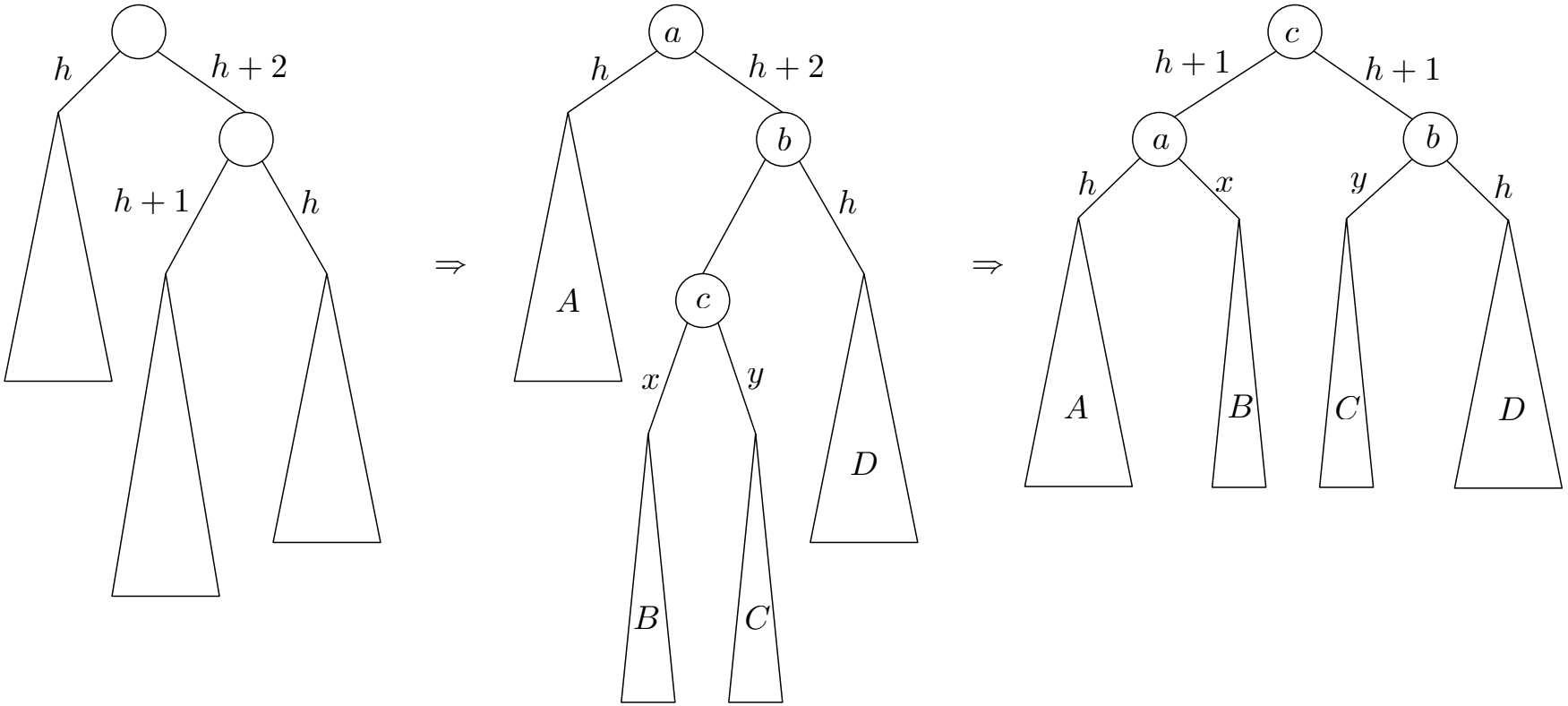
Similar to Left-Left, fix by a rotation:



Note that $x = h$ or $h + 1$, and the ordering from left to right of A, a, B, b, C is preserved after rotation.

Right-Left

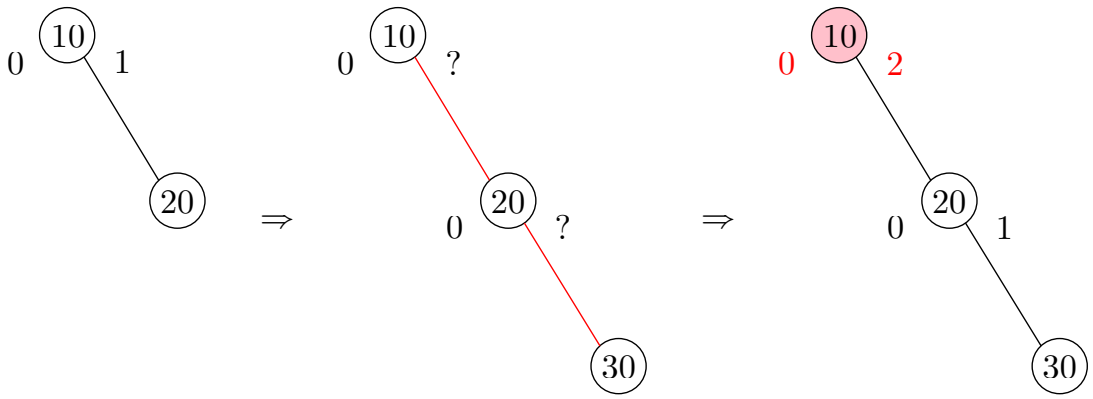
Similar to Left-Right, fix by a double rotation:



Note that x and y must be h or $h-1$. Futhermore at least one of them must be h.

Insertion

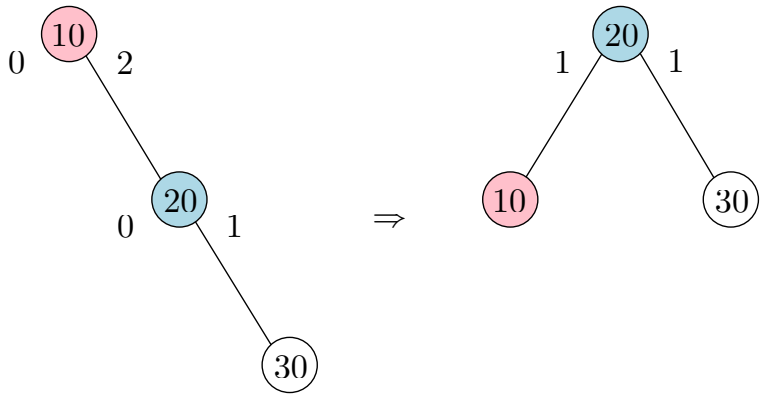
Let's see these in action with some concrete examples involving insertion and deletion. Suppose we start with an empty tree and add 10, 20 and 30. Inserting 30 yields:



We first traverse from root-to-leaf and add a node with key 30. The height of the subtrees along this path are now invalidated, so we traverse back up to the root and recalculate at each node. When we get to node 10, we find that we have an imbalance, in this case of type Right-Right.

Insertion

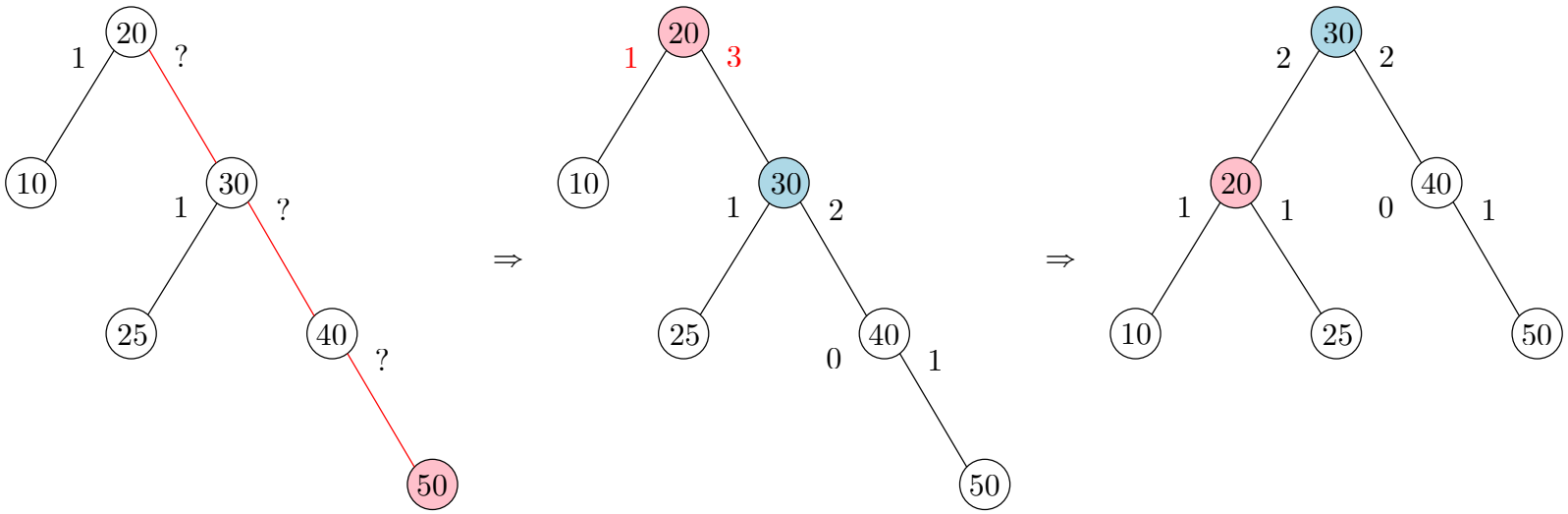
We fix via a rotation according to the Right-Right case:



One should check that at the end the tree is balanced, and satisfies the binary search tree property!

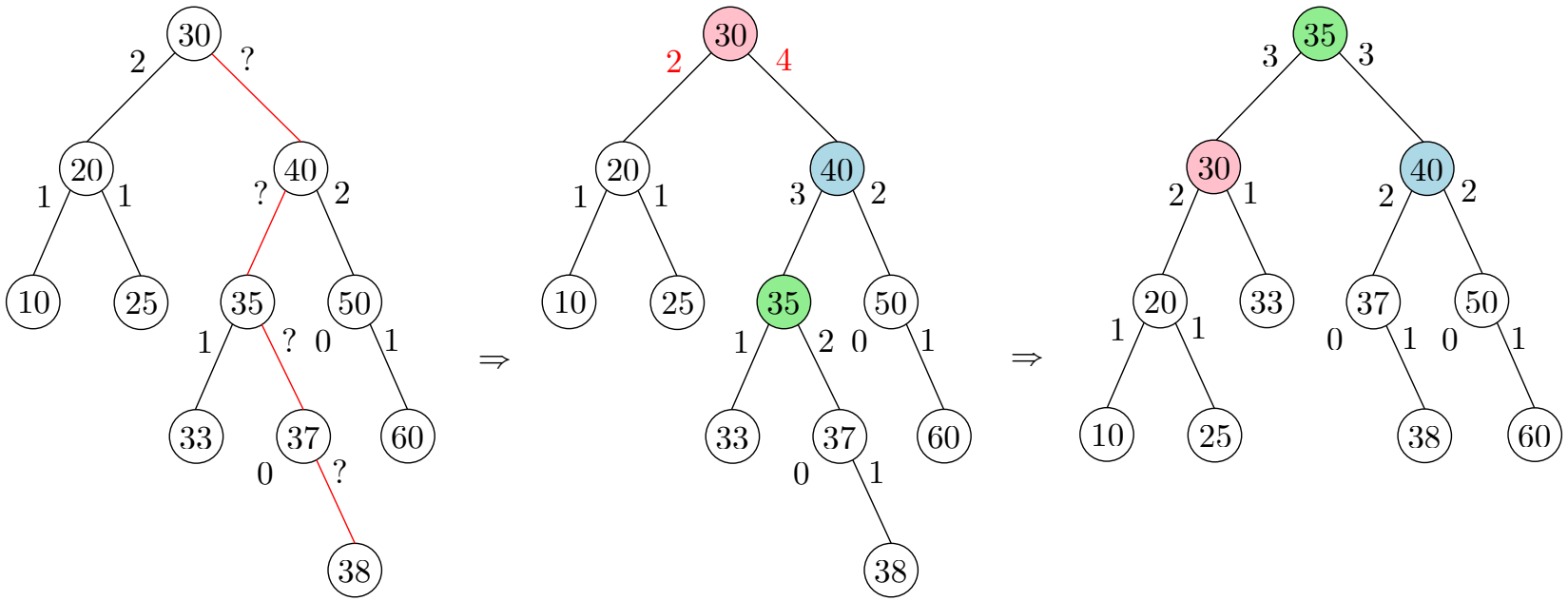
Insertion

Suppose we then add 25, 40 and 50. Upon inserting 50, we will find the need for another Right-Right rebalancing:



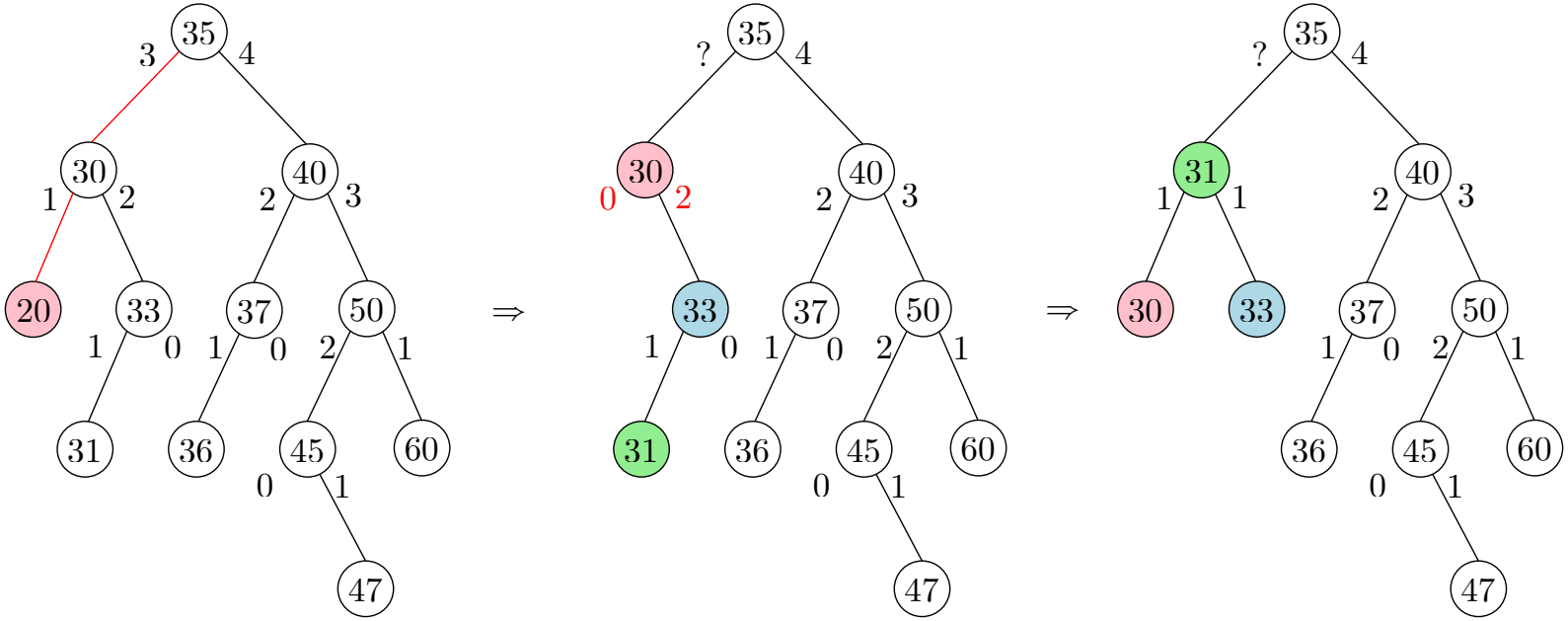
Insertion

To get a Right-Left case, let's add 35, 33, 37, 60 and 38 (in this exact order). Upon inserting 38 we will find:



Deletion

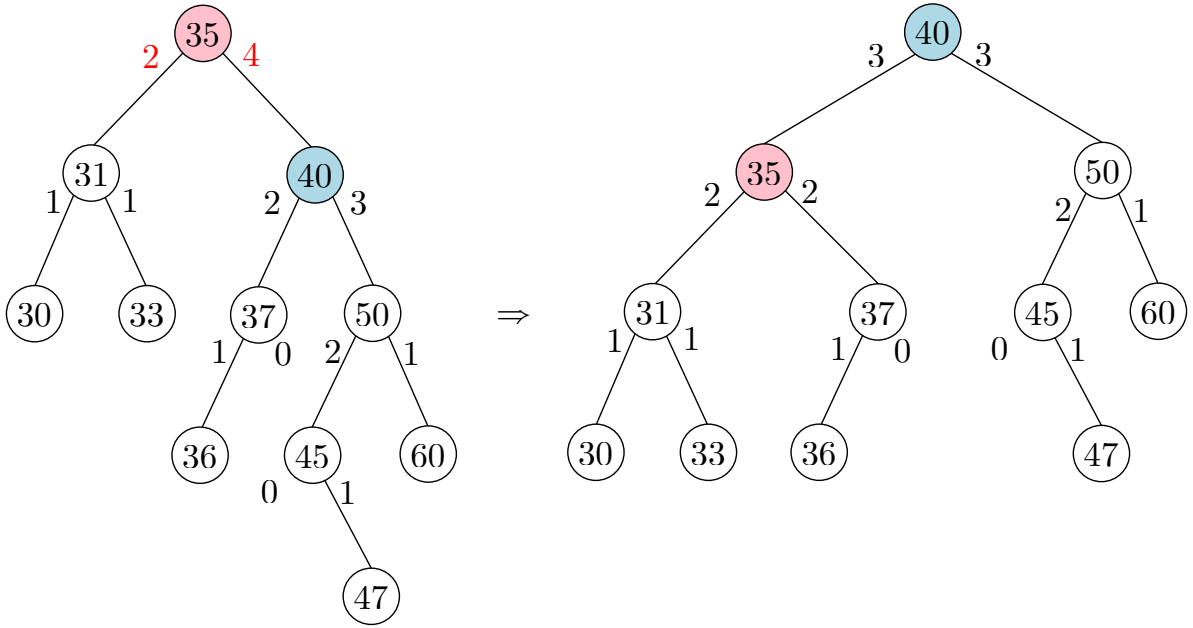
Let's look at an example of a deletion as well: suppose that some sequence of operations later we arrive at the following and want to delete the key 20:



After identifying and deleting the appropriate node (more on this later), we again need to traverse back up to the root and rebalance. Here we run into a Right-Left case.

Deletion

. . . and we are actually still not done because there is another Right-Right rebalance we need to do:

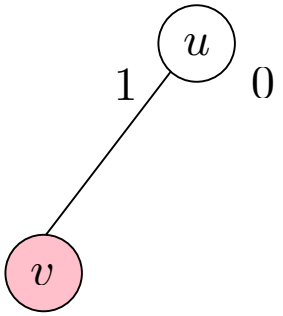


Remember that at most one rebalance is needed on insert; but deletion may require more than one!

Deletion

This example of deletion happened to be easy because the node holding the key to be removed was a leaf node. Identifying which node to delete can be tricky if the key is at an internal node.

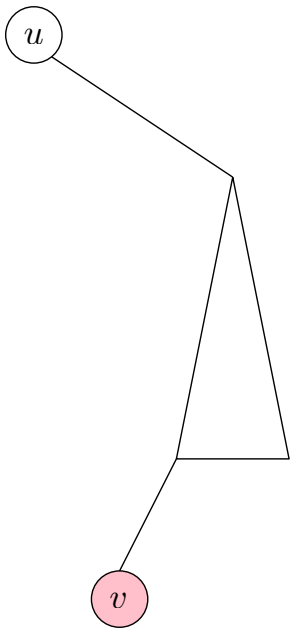
Suppose that the node holding the key we want to remove is an internal node named u. Recall that the idea is based on identifying the successor node. As a base case, if the right subtree is empty then we don't have a successor. This means we must have a single left child that is a leaf (because the tree is balanced):



In this case we swap the keys of u and v and v is the node we delete. (This corresponds to Case 3 from the lecture notes).

Deletion

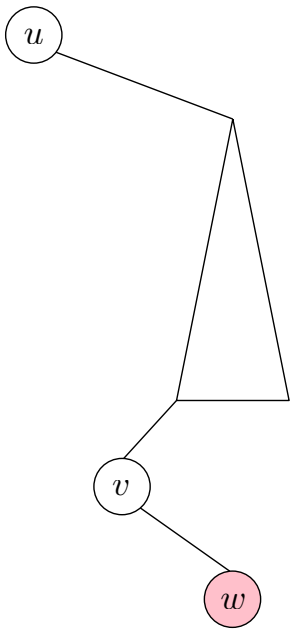
If we do have a right subtree, then a successor exists. We can find it in exactly the same way as our successor query by considering the subtree rooted at u. Let's call it v. After we've found the successor, if it is a leaf node:



We swap keys between u and v and delete v. (This is Case 2.1).

Deletion

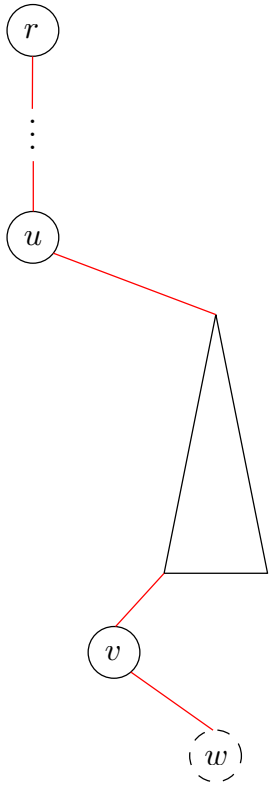
. . . however the successor may not be a leaf node. We know that its left subtree must be empty (else there would be a node with key between those of u and v, contradicting v being the successor), and so the right child must be a leaf node w:



Notice that w is the successor of v here, and we copy v's key into u and w's key into v and remove w. (This is Case 2.2).

Deletion

The nodes we need to check for possible imbalances when we run `delete` are all the ones on the path from the leaf node we actually remove up to the root node:



In the last most complicated case we considered, u is the node holding the key we wish to remove and w is the node we actually remove. The leaf-to-root path we traverse (and check) starts at w not u!