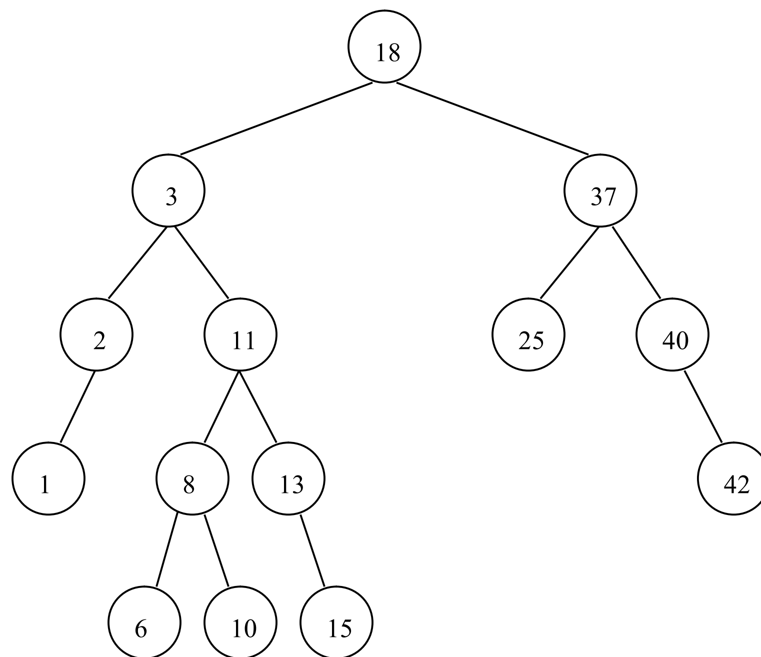
AVL trees
Notes by Clark Olson and Carol Zander

An AVL tree must have the following properties:

- It is a binary search tree.
- For each node in the tree, the height of the left subtree and the height of the right subtree differ by at most one (the balance property).

The height of each node is stored in the node to facilitate determining whether this is the case. The height of an AVL tree is logarithmic in the number of nodes. This allows insert/delete/retrieve to all be performed in O(log n) time. Here is an example of an AVL tree:



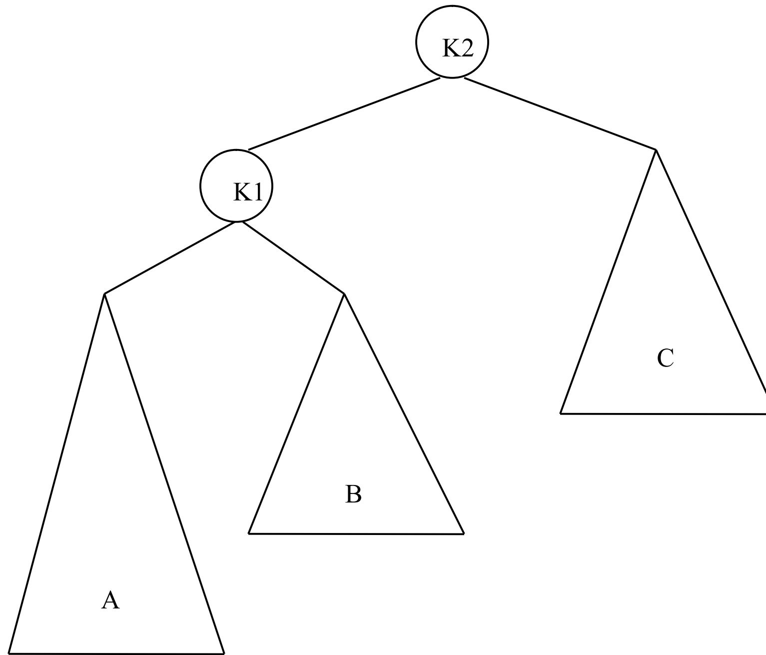
Inserting 0 or 5 or 16 or 43 would result in an unbalanced tree.

The key to an AVL tree is keeping it balanced when an insert or delete operation is performed. If we start with an AVL tree, then what is needed is either a single rotation or a double rotation (which is two single rotations) on the unbalanced node and that will always restore the balance property in O(1) time. Note that the rotations are always applied at the lowest (deepest) unbalanced node and no nodes above it need to have rotations applied.
When a node becomes unbalanced, four cases need to be considered:

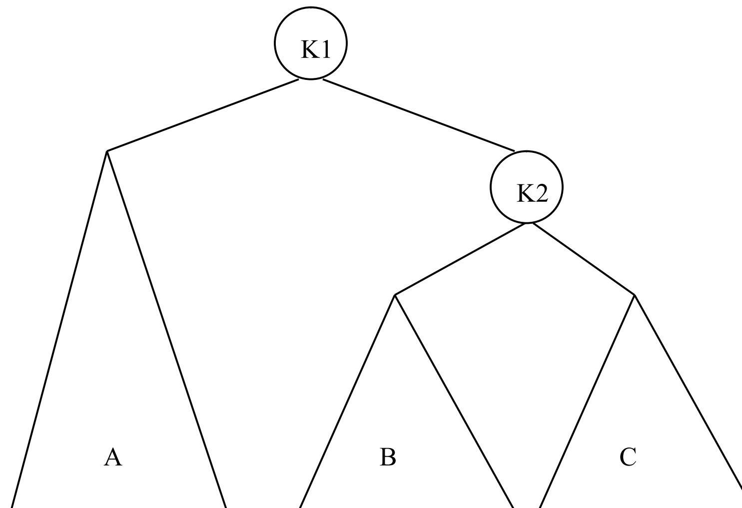
- Left-left: The insertion was in the left subtree of the left child of the unbalanced node.
- Right-right: The insertion was in the right subtree of the right child of the unbalanced node.
- Left-right: The insertion was in the right subtree of the left child of the unbalanced node.
- Right-left: The insertion was in the left subtree of the right child of the unbalanced node.

Single rotations

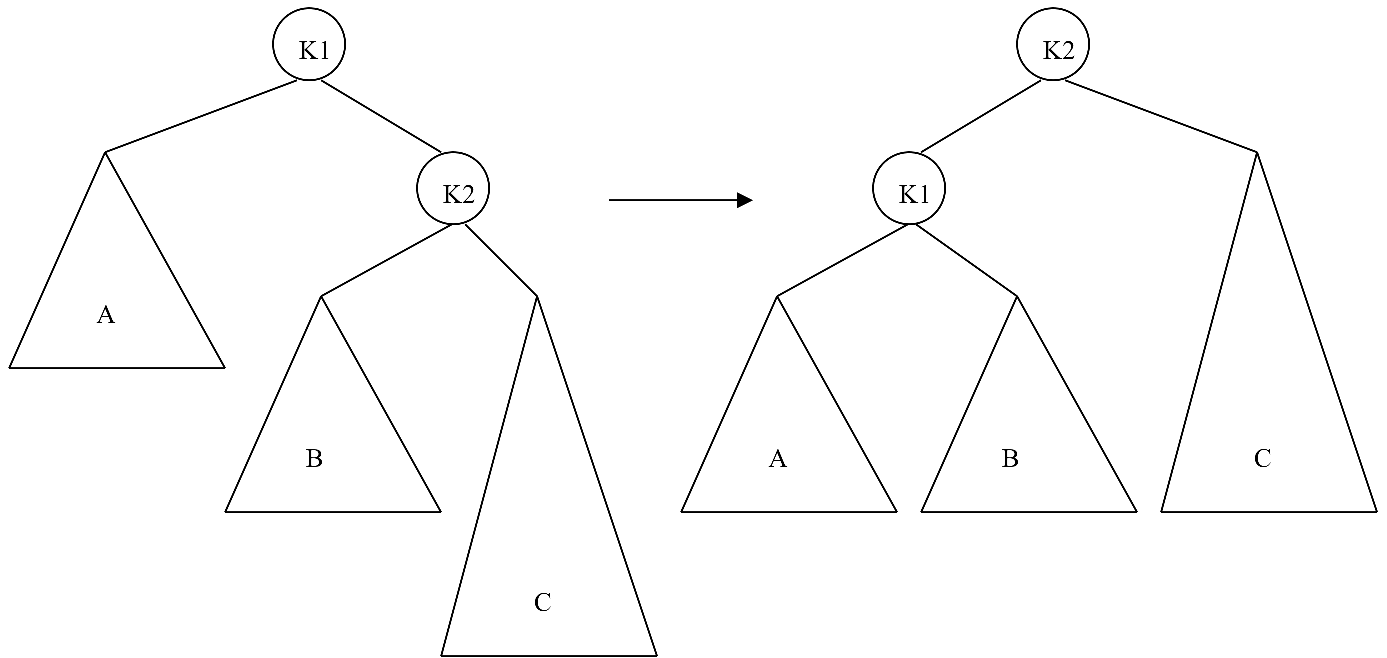
The first two require single rotations and the last two require double rotations to rebalance the tree. In general, a left-left looks like this:



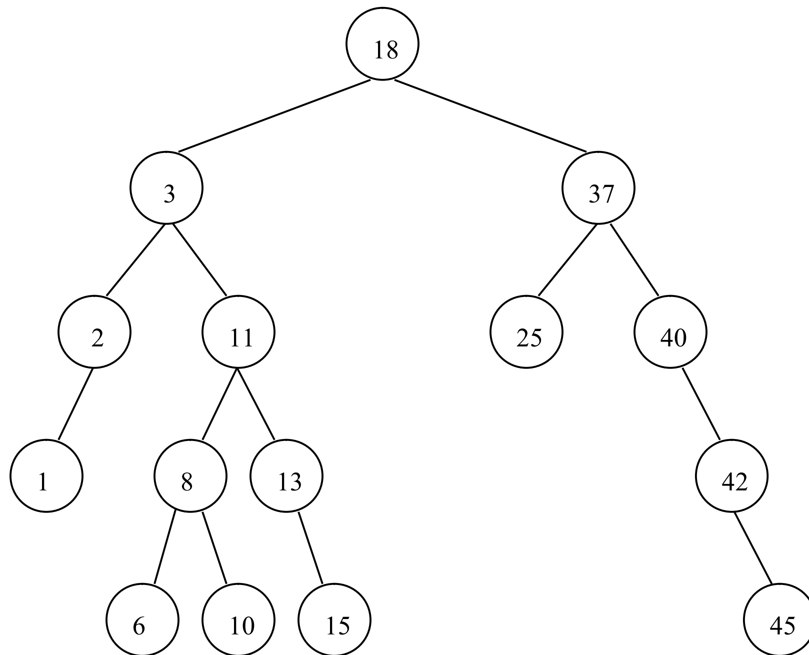
K2 is the unbalanced node. Nothing above it matters. The left-left subtree A always has height one greater than the right subtree C, which unbalances the tree. The solution is to make K1 the root of this (sub)tree with K2 as its' right child and B as the right-left subtree:



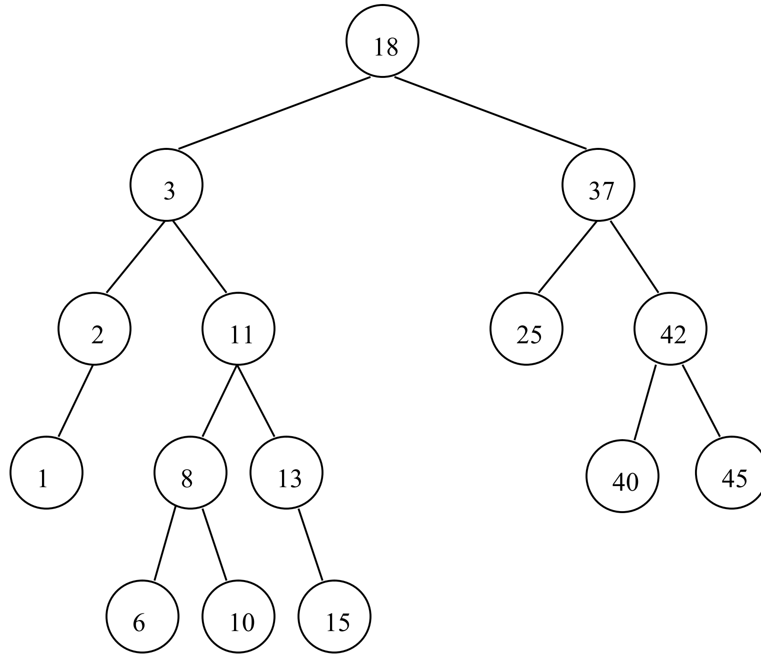
A right-right is symmetrical.



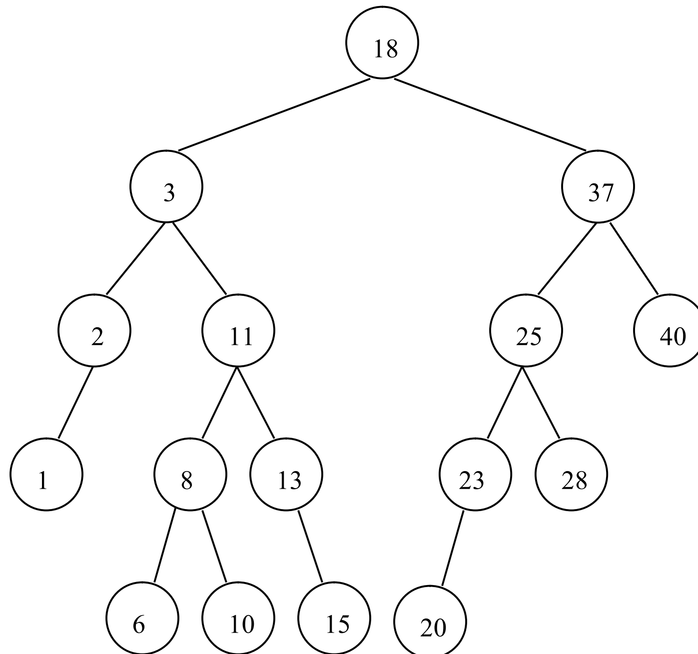
Here is a right-right example with actual data:



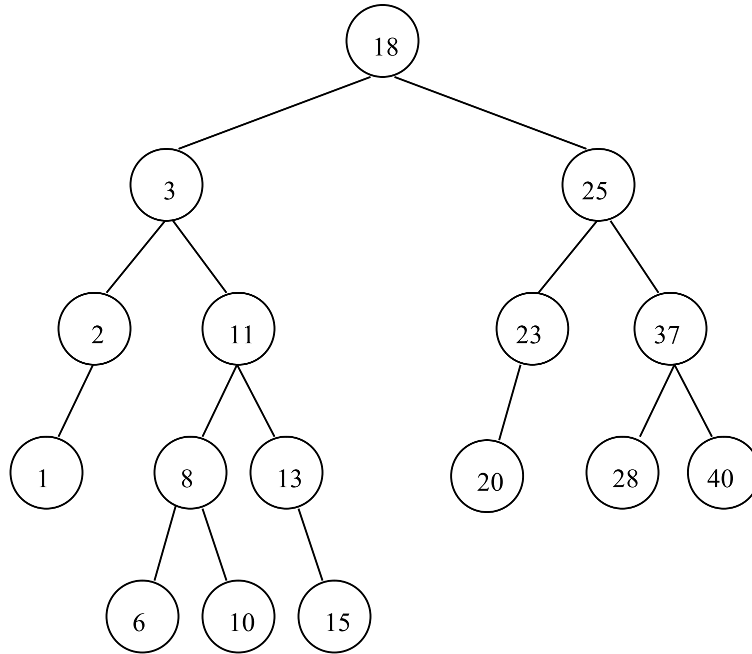
Nodes 37 and 40 are unbalanced. The lower node needs to be fixed. This is a right-right, since 42 is the right child of 40 and 45 is in the right subtree of 42. In this case, two of the subtrees, A and B, are empty. The subtree C has the single node 45. Rotating (with right child) yields:



Here is a left-left, where the unbalanced node is not as deep:

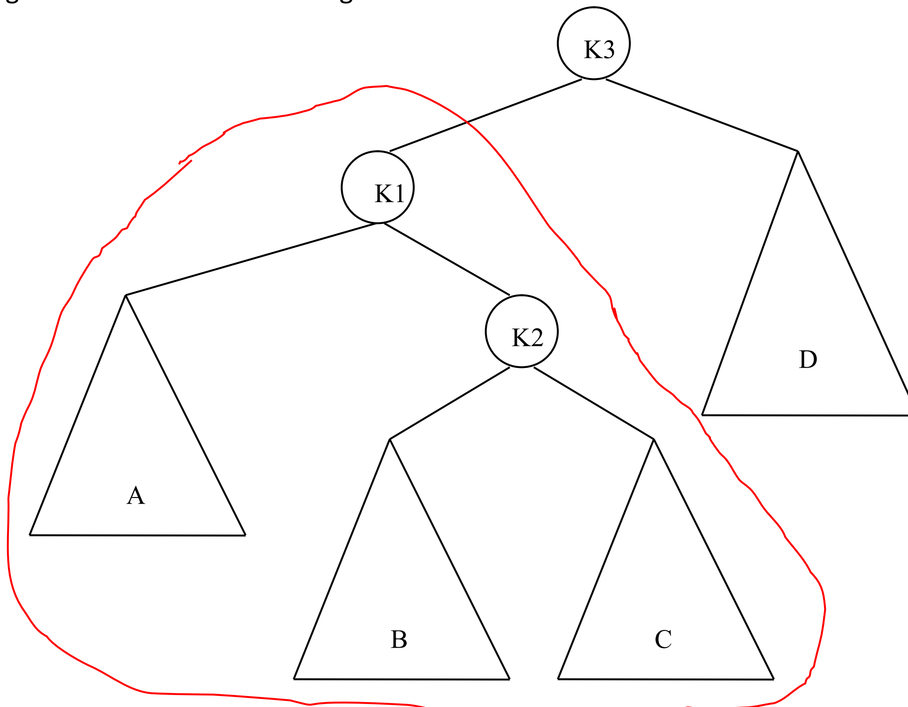


Node 37 is unbalanced. The inserted node is in the left subtree of the left child of 37. Therefore, we rotate 37 with its' left child:

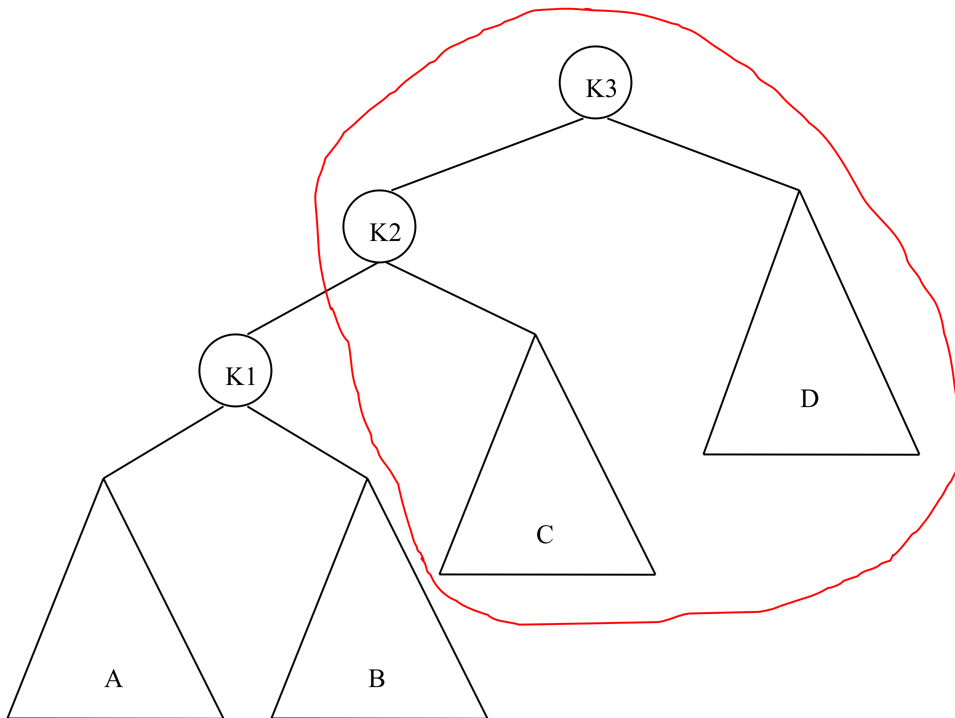


Double rotations

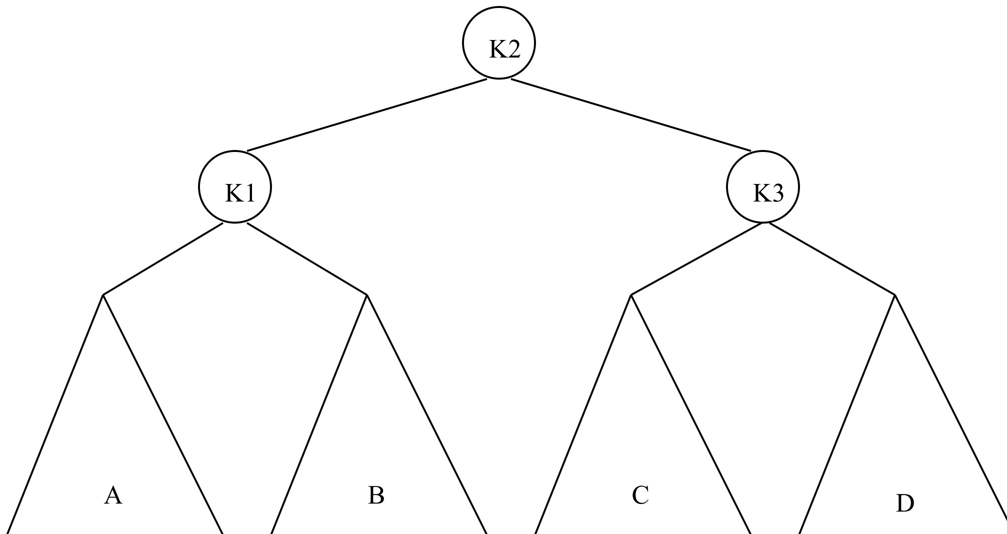
When we have the left-right and right-left cases, a single rotation is not sufficient to rebalance the tree. In this case, we need to perform a double rotation, which consists of two single rotations. Here is the generic situation for a left-right:



To fix this, we need to rotate K2 all the way to the top. First, we do a right-right rotation of K1 with K2 and then a left-left rotation of K3 with K2. The first rotation (K1-K2) yields:

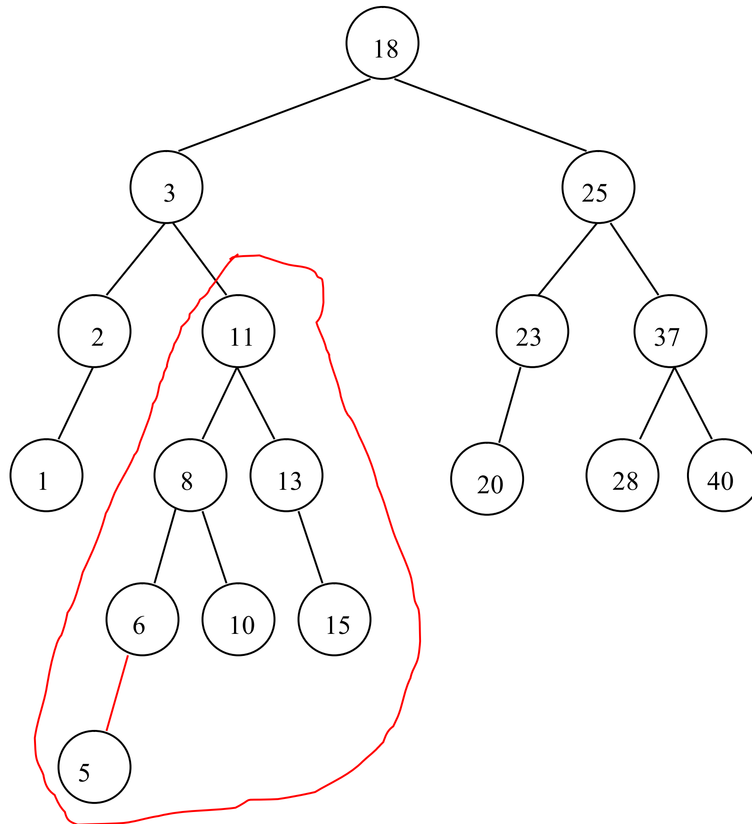


The second rotation (K2-K3) gives us:

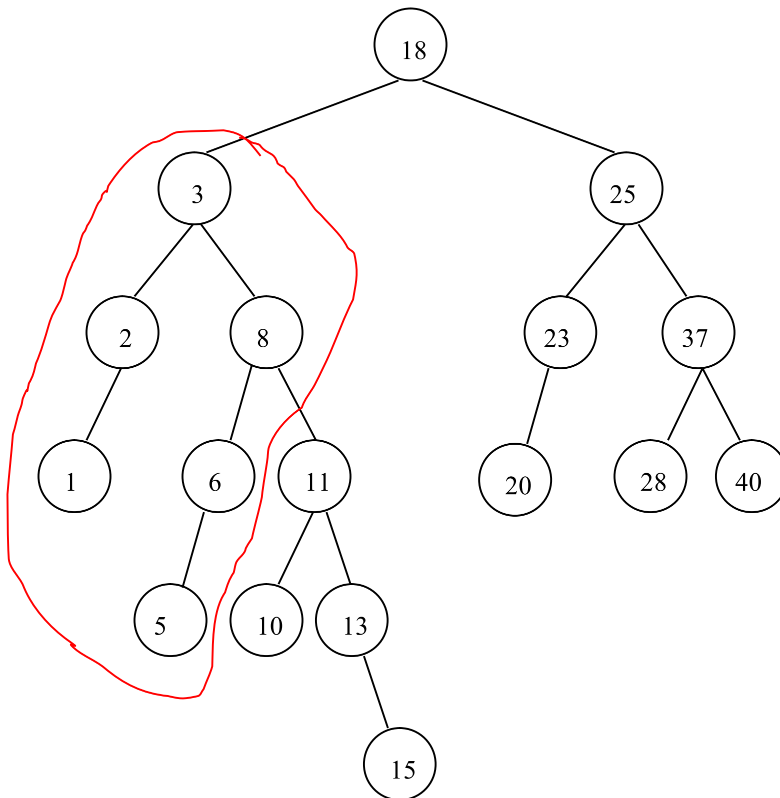


Conceptually, the K2 “splits” the K1 and the K3 and rises to the top, with the K1 and the K3 ending up being children and A, B, C, and D being grandchild trees.

A right-left rotation is a mirror image of left-right. To end up with the same final structure, let k1 be the root of the unbalanced tree. Node k3 is its right child and k2 is the left child of k3. First, do a left-left on tree k3-k2. Then do a right-right on tree k1-k2. Here is an example of a right-left with real data. It is an AVL tree until the 5 is inserted which make it unbalanced at k1= 3.



First rotation (unbalanced at k1 = 3) (do left-left on k3 = 11, k2 = 8):



Second rotation (do right-right on k1= 3, k2 = 8):