CSS 343 – Data Structures, Algorithms, and Discrete Mathematics II
Professor Clark F. Olson
Lecture Notes 9 – Balanced search trees

Reading: Carrano, 19.1, 19.5

AVL trees

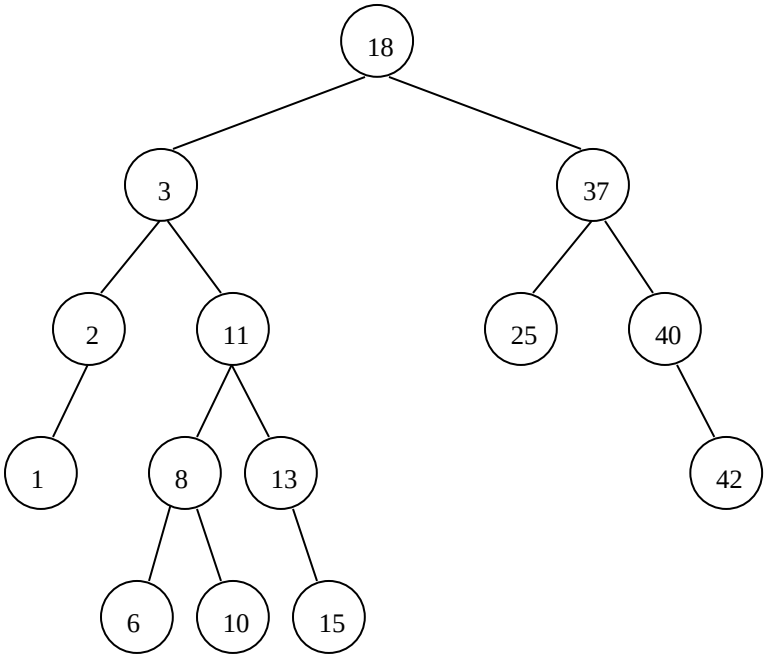
One potential problem with an ordinary binary search tree is that it can have a height that is O(n), where n is the number of items stored in the tree. This occurs when the items are inserted in (nearly) sorted order. We can fix this problem if we can enforce that the tree remains balanced while still inserting and deleting items in O(log n) time. The first (and simplest) data structure to be discovered for which this could be achieved is the AVL tree, which is names after the two Russians who discovered them, Georgy Adelson-Velskii and Yevgeniy Landis. It takes longer (on average) to insert and delete in an AVL tree, since the tree must remain balanced, but it is faster (on average) to retrieve.

An AVL tree must have the following properties:

- It is a binary search tree.
- For each vertex 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 vertex is stored in the vertex to facilitate determining whether this is the case. Interestingly, the minimum number of vertices in an AVL tree with height h is one less than the (h+3) Fibonacci number. (The proof uses induction.) Since the Fibonacci numbers grow exponentially, this implies that the height of an AVL tree must be logarithmic in the number of vertices. 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. We will examine insertion in detail. If an insertion causes the balance property to be violated, then we must perform a tree "rotation" in order to fix the balance. 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 vertex and that will always restore the balance property in

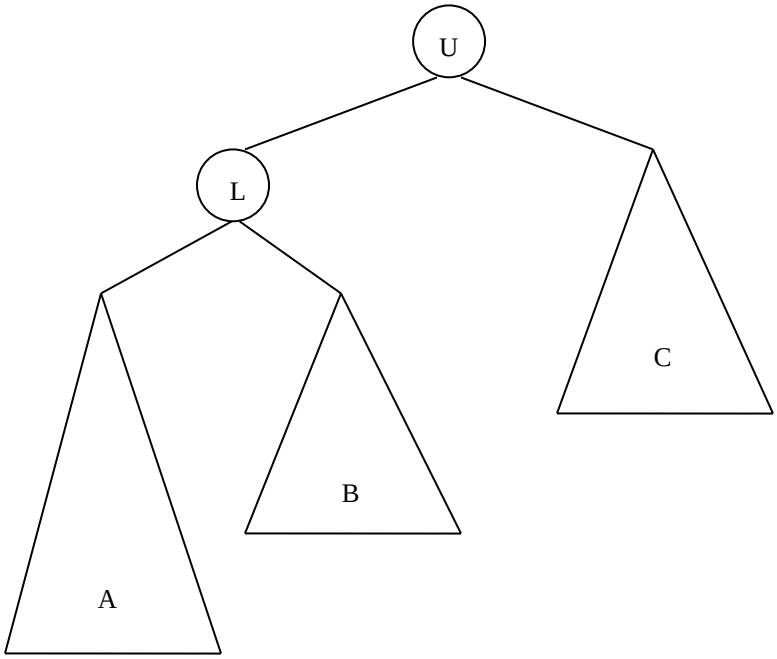
O(1) time. Note that the rotations are always applied at the lowest (deepest) unbalanced vertex and no vertices above it need to have rotations applied.

When a vertex becomes unbalanced, four cases need to be considered:

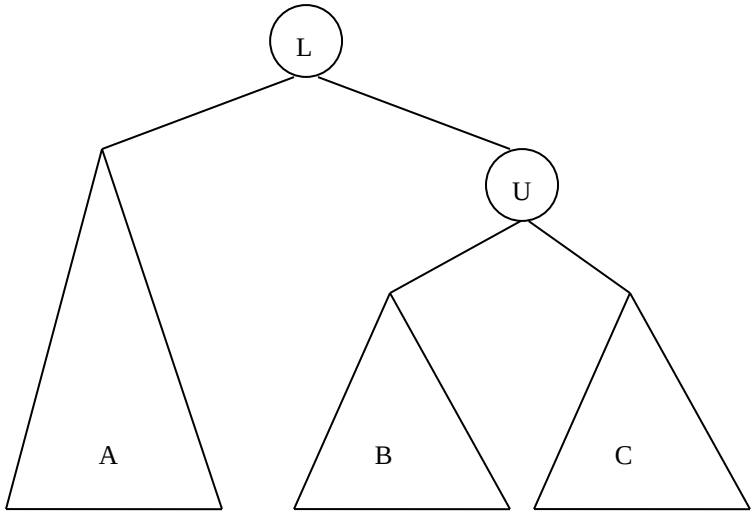
- Left-left: The insertion was in the left subtree of the left child of the unbalanced vertex.
- Right-right: The insertion was in the right subtree of the right child of the unbalanced vertex.
- Left-right: The insertion was in the right subtree of the left child of the unbalanced vertex.
- Right-left: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

Single rotations

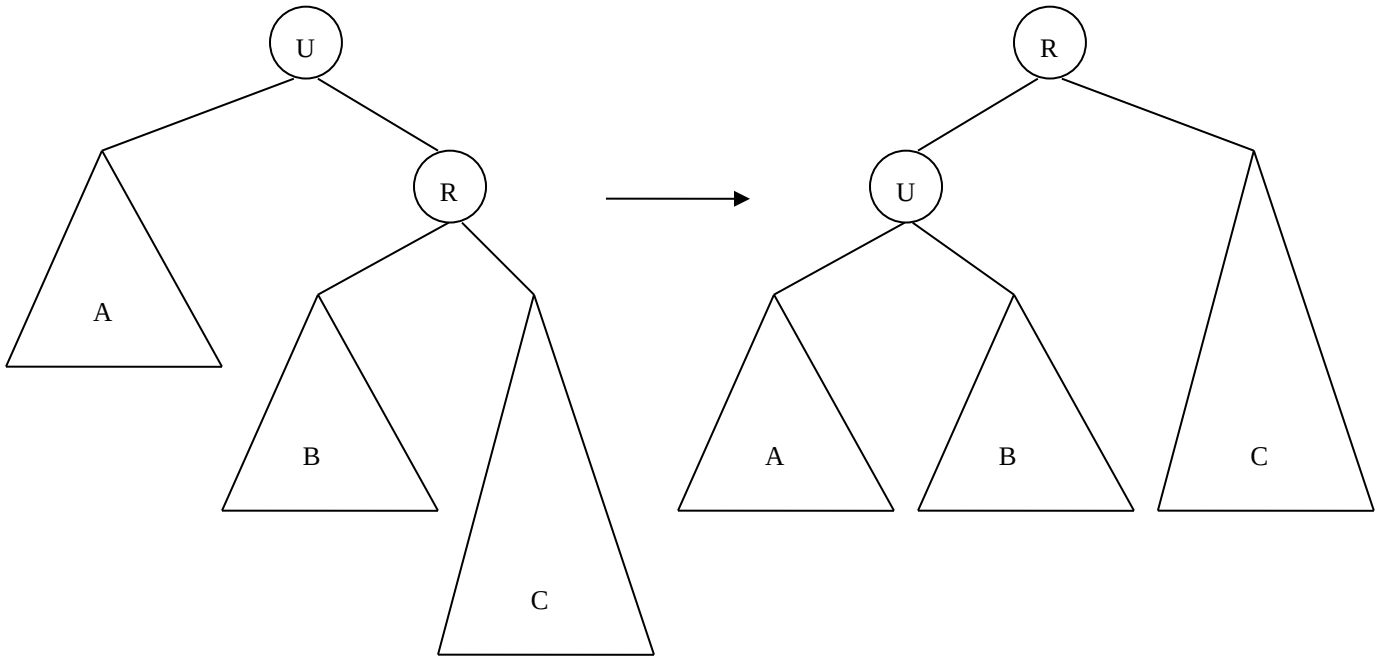
The first two require single rotations and the last two require double rotations to rebalance the tree. In general, a left-left looks like this:



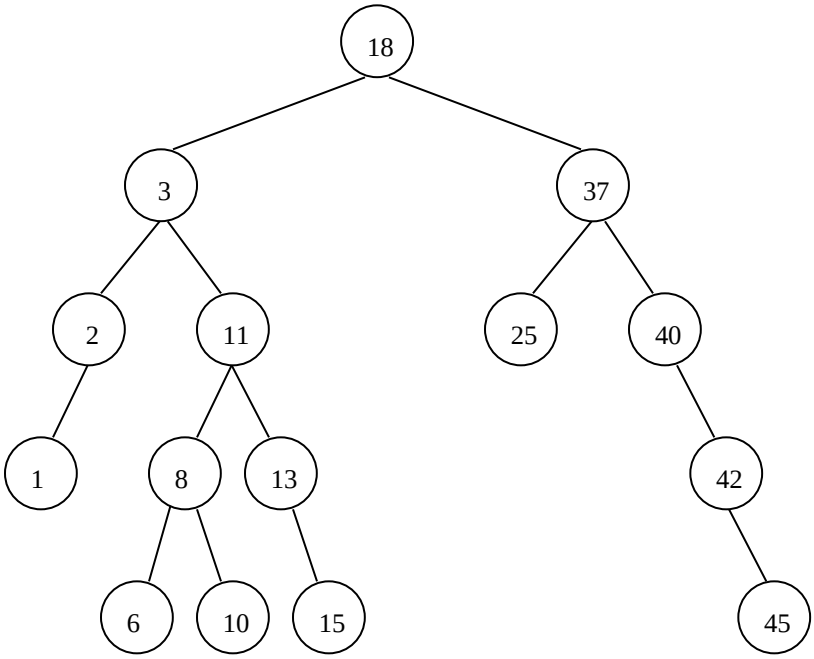
U is the unbalanced vertex. Nothing above it matters. The left-left subtree A always has height one greater than the right subtree C, which unbalances the tree. The solution is to make L the root of this (sub)tree with U as its' right child and B as the right-left subtree:



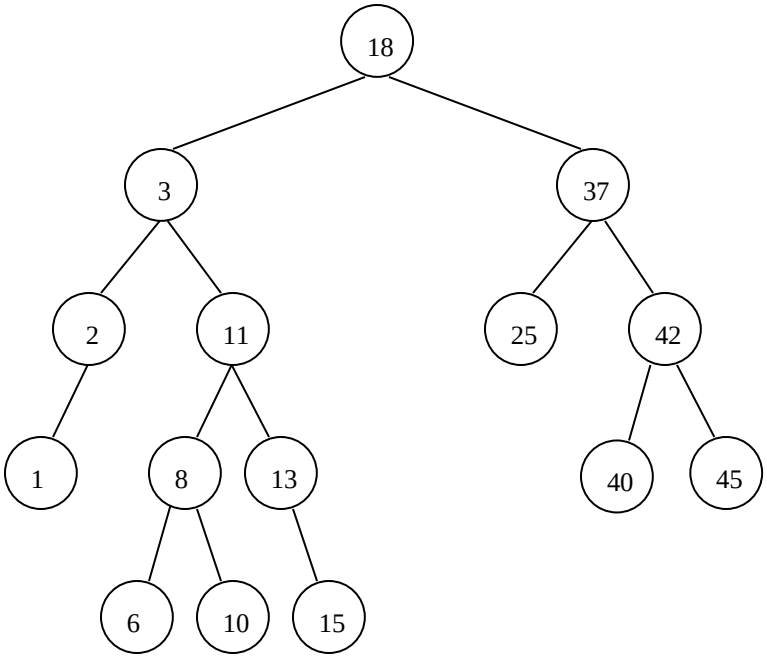
A right-right is symmetrical.



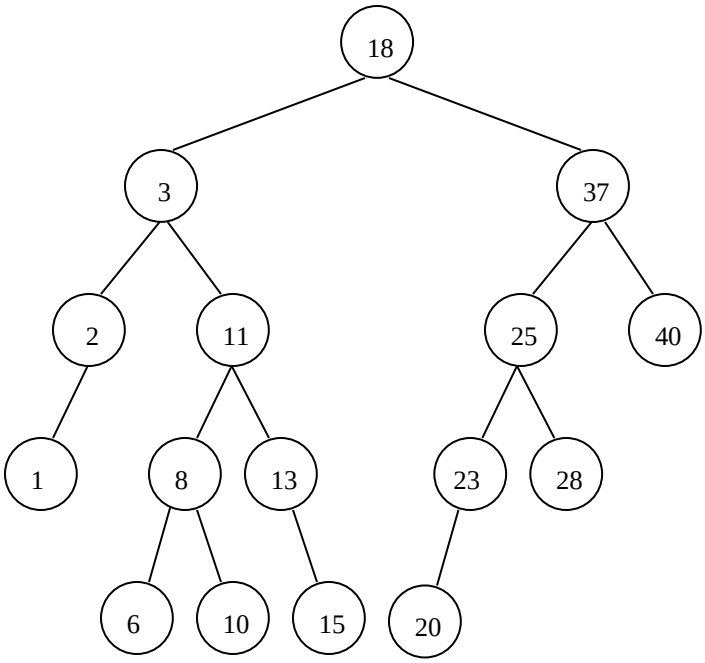
Here is a right-right example with actual data:



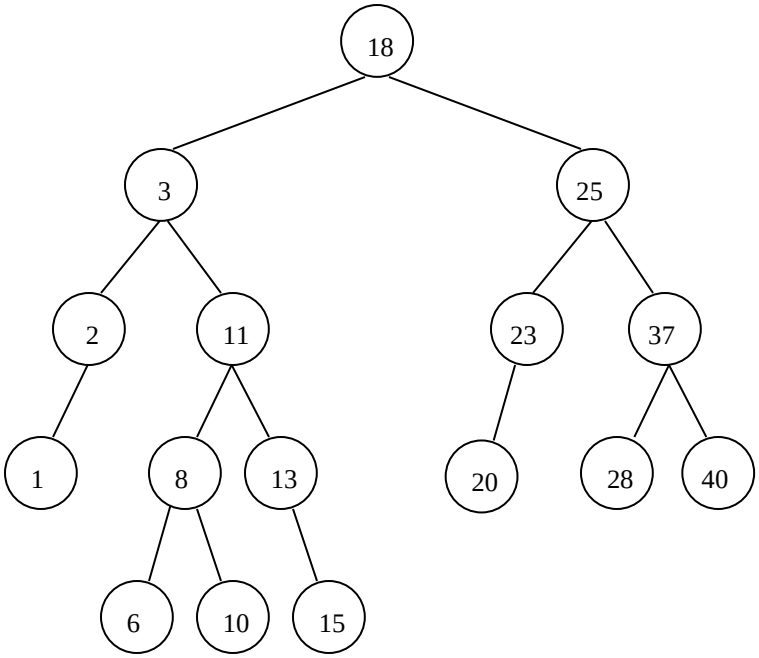
Vertices 37 and 40 are unbalanced. The **lower** vertex needs to be fixed. This is a right-right, since 42 is the right child of 40 and 45 is in the right subtree of 42. In this case, two of the subtrees, A and B, are empty. The subtree C has the single vertex 45. Rotating (with right child) yields:



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



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

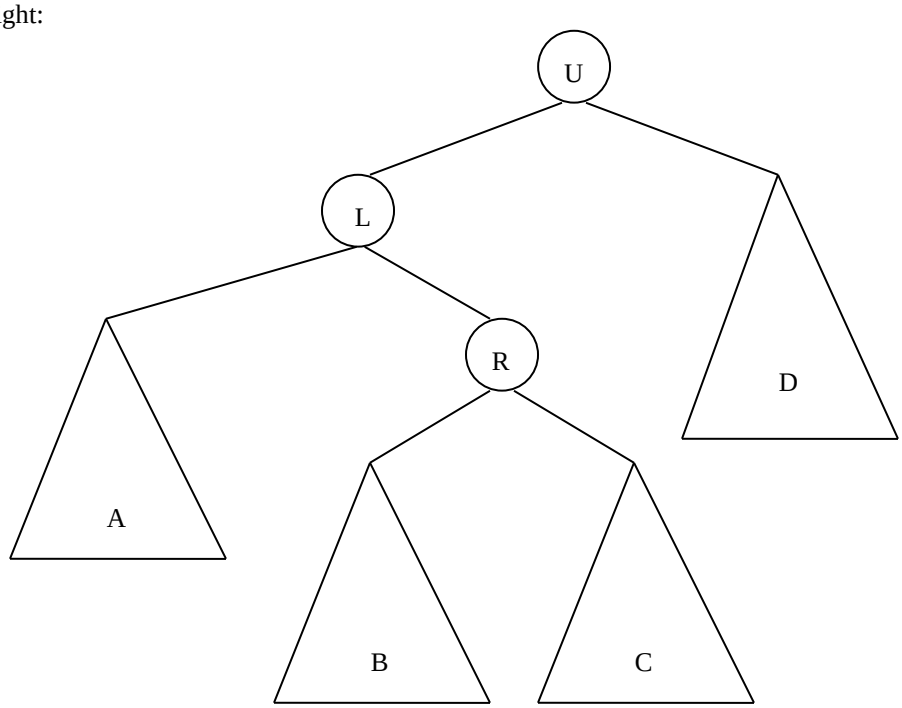


The code for a single rotation is actually very short:

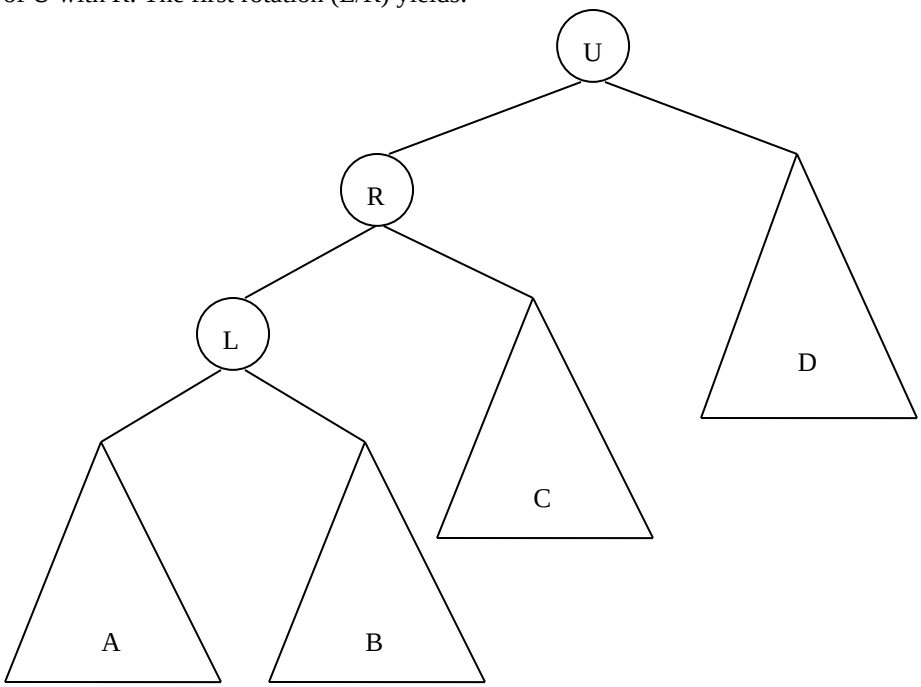
```
void BSTree::rotateWithLeftChild(Node *&U)
{
        Node *L = U->leftChild;
        U->leftChild = L->rightChild;
        L->rightChild = U;
        U = L;
}
```

Double rotations

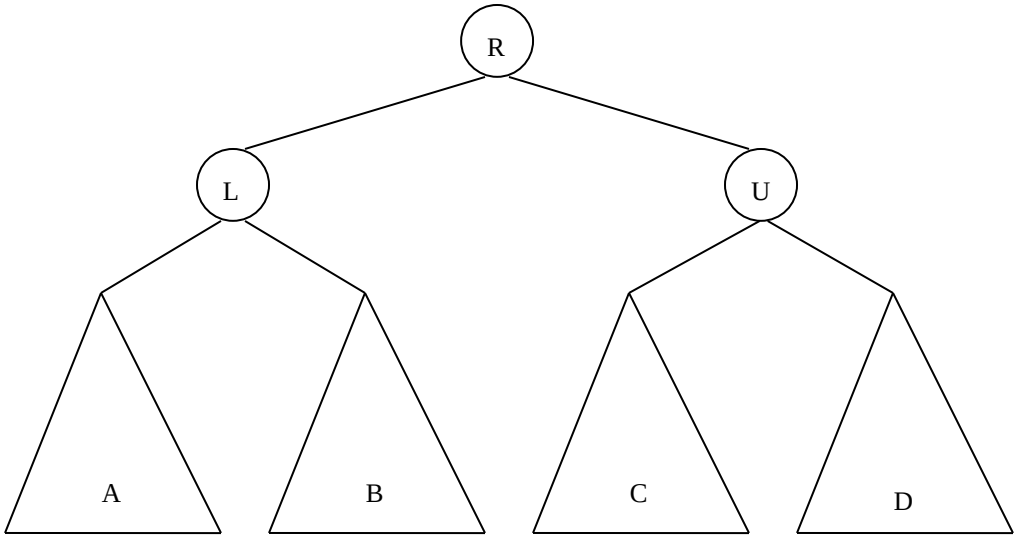
When we have the left-right and right-left cases, a single rotation is not sufficient to rebalance the tree. In this case, we need to perform a double rotation, which consists of two single rotations. Here is the generic situation for a left-right:



To fix this, we need to rotate R all the way to the top. First, we do a right rotation of L with R and then a left rotation of U with R. The first rotation (L/R) yields:

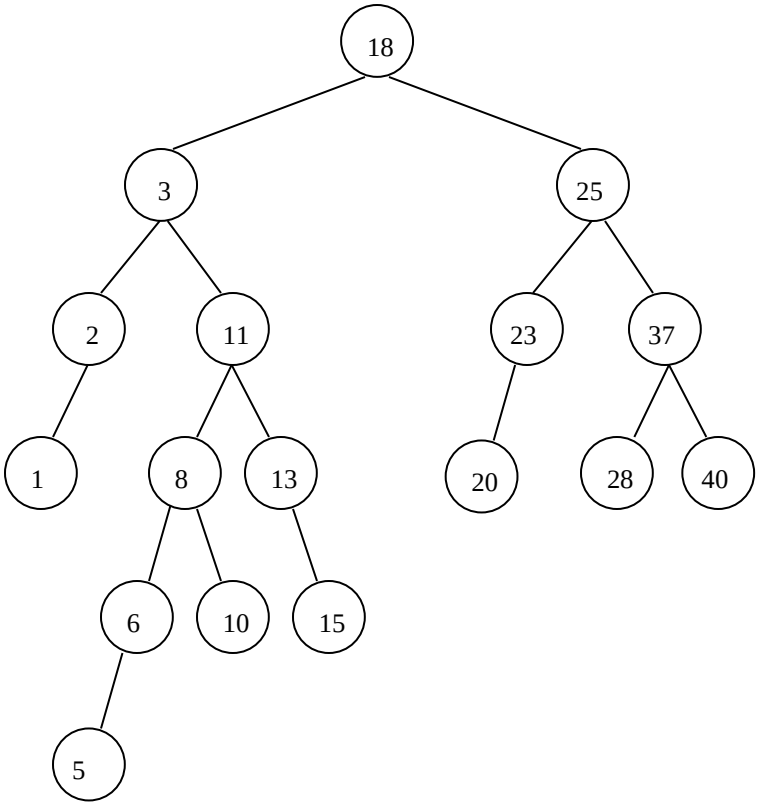


The second rotation (R/U) gives us:

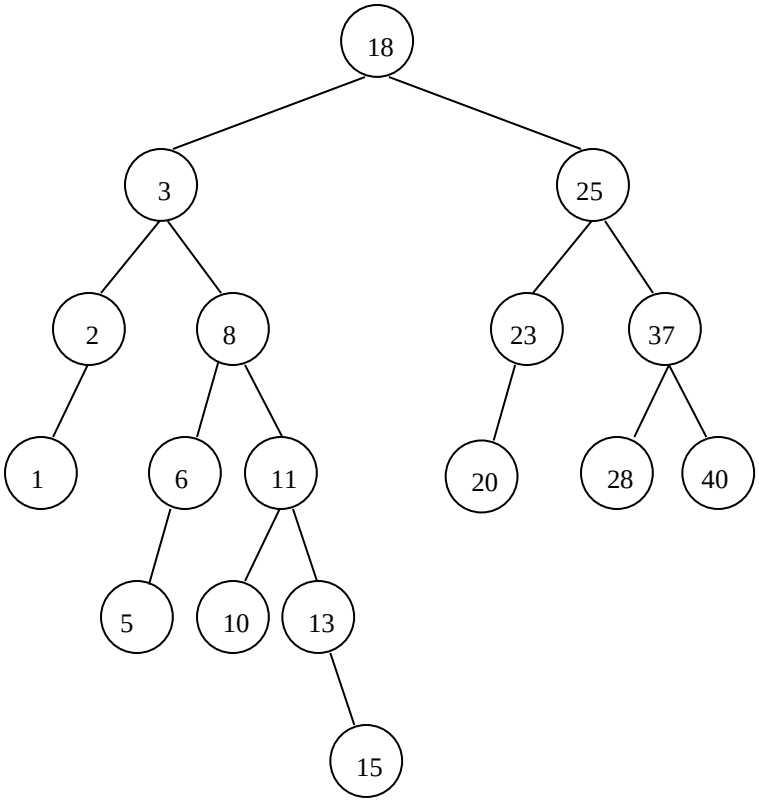


Conceptually, the R "splits" the L and the U and rises to the top, with the L and the U being subchildren and A, B, C, and D being grandchild trees.

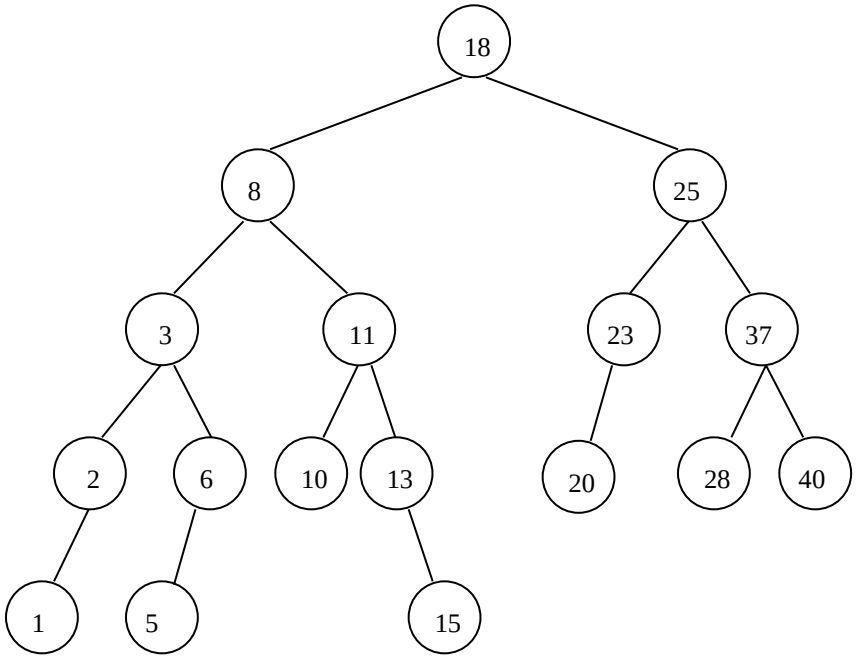
A **right-left** case is symmetrical. Here is an example of a right-left with real data:



The vertex storing 3 is unbalanced. This is a right-left, since 11 is a right child and 8 is a left child. The first rotation rotates 8 with 11:



The second rotation rotates 3 with 8:



The code for a double rotation is even simpler than for a single rotation (if you already have the single rotation code):

```
void BSTree::doubleRotateWithLeftChild(Node *&U) {
        rotateWithRightChild(U->leftChild);
        rotateWithLeftChild(U);
}
```

Implementation, delete, and summary

The implementation of insert in AVL requires both a top-down pass and a bottom-up pass through the tree. In the top down pass, the correct location for the new object is found and it is inserted. The tree is then traversed back towards the root in order to maintain the height of each node and perform the balancing operation. This is done after the recursive call in the method implementation.

In order to perform delete, we have to combine the code for the delete in a simple (unbalanced) binary search tree with the rebalancing that we perform upon an insert.

AVL trees generally maintain better balance than some other balanced tree structures, but require more rotations in order to keep this balance. They are used in practice when retrieves are common, but not inserts and deletes. Other (more complex) balanced binary trees perform a bit better when insert and deletes are common. The rotations in AVL trees demonstrate important concepts that are used in other balanced trees. In my opinion, the more complex balanced trees yield a modest improvement (a better constant factor).

Red-black trees

A balanced binary search tree that can be implemented to run a bit faster than an AVL tree is the red-black tree that has the following properties:

- It is a binary search tree.
- Each node is colored red or black (the colors aren't important, but these are used for historical reasons).
- The root is black.
- If a node is red, its children must be black (you can't have two consecutive red nodes).
- Each path from a node to any descendant leaf of that node contains the same number of black nodes.

Together, these properties ensure that the tree is reasonably balanced. (This is because there can't be consecutive red nodes and all of the paths from the root to any of the leaves must contain the same number of black nodes.) The maximum height turns out to be 2 log (n+1).

In implementing red-black trees, there are more cases that need to be dealt with. We won't consider them in detail, but they all require single or double rotations, combined with recoloring of some nodes. However, these can be implemented with only a top-down traversal of the tree and, thus, don't need recursion. They are tricky to implement owing to number of cases for rotations and special cases with respect to the root and empty subtrees.

AA-trees

AA-trees are a special type of red-black trees in which the left child of a node can never be red. This actually simplifies the implementation, in part because it removes half of the cases that can arise. Implementations are still based on rotations and recoloring.

Practice problems (optional):

Carrano, Chapter 19:

#1a, e

Insert the following numbers (in order) into an AVL tree: 5 1 7 9 13 3 4 10 16 19 [single rotations]

Insert the following numbers (in order) into an AVL tree: 25 14 31 11 15 21 17 27 33 26 [double rotations]