AVL trees

Mikael Tylmad

March 5, 2008

1 Introduction

A binary tree is a very effective and simple data structure. Inserting and deleting values is easy and finding values only takes $\log(n)$ operations. There is however a large downside to all this: unbalance. If the elements that are inserted are already in order, the binary tree becomes unbalanced and works no better than a simple linked structure. A solution to this is to implement a self-balancing binary search tree. An AVL tree is just that. In 1962 G.M. Adelson-Velsky and E.M. Landis published a paper called “An algorithm for the organization of information”. In this paper they introduced the first self-balancing tree which was later named after its inventors: the AVL tree.

In an AVL tree each node keeps track of the height of its two subtrees. A balance factor can then be calculated by taking the height of the right subtree minus the height of the left subtree. If an unbalance is detected, the tree performs re-balancing operations. Lookup, insertion and deletion in an AVL tree takes $\Theta(\log(n))$ operations in average and worst case scenarios alike.

2 Implementing the AVL tree

An AVL tree requires partly special operations, shown in section 2.2, and extra data in each node. This data consists of references to the children of the node, its parent and exactly how high its subtrees are. An example of this can be seen in figure 1.

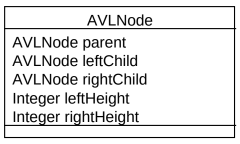


Figure 1: An example of how each node could be implemented

2.1 Common operations

There are three common operations: insertion, deletion and lookup. Insertion and deletion are connected as they both work with the balance factor of each node and that they use the special tree operations explained in section 2.2. Lookup is performed exactly as in a normal binary search tree.

2.1.1 Insertion

Inserting values in an AVL tree is at first analogous to inserting values in a binary search tree. When the proper place for an element has been found and the element has been inserted, the balancing process begins. Beginning at the inserted element, the path to the root is retraced. At each element the height of that elements right or left subtree is increased by one depending from where the retracing came from. If the difference between the height of the left subtree and the right subtree is -1, 0 or 1 the tree rooted at this element is still in perfect shape. In that case the retracing towards the root can continue. However if the difference is -2 or 2 the tree rooted at this element is unbalanced. To correct this one or two tree rotations is needed. There are 4 cases that need to be accounted for. For simplicity the current root will be called *Ro*, the roots right child *RoRC* and the roots left child *RoLC*.

1. If the balance factor of *Ro* is 2 and the balance factor of *RoRC* is 1, a left rotation is needed with *Ro* as the pivot point.
2. If the balance factor of *Ro* is -2 and the balance factor of *RoLC* is -1, a right rotation is needed with *Ro* as the pivot point.
3. If the balance factor of *Ro* is 2 and the balance factor of *RoRC* is -1, two rotations are needed. First a right rotation is with *RoRC* as the pivot point and then a left rotation with *Ro* as the pivot point.
4. If the balance factor of *Ro* is -2 and the balance factor of *RoLC* is 1, two rotations are needed. First a left rotation is with *RoLC* as the pivot point and then a right rotation with *Ro* as the pivot point.

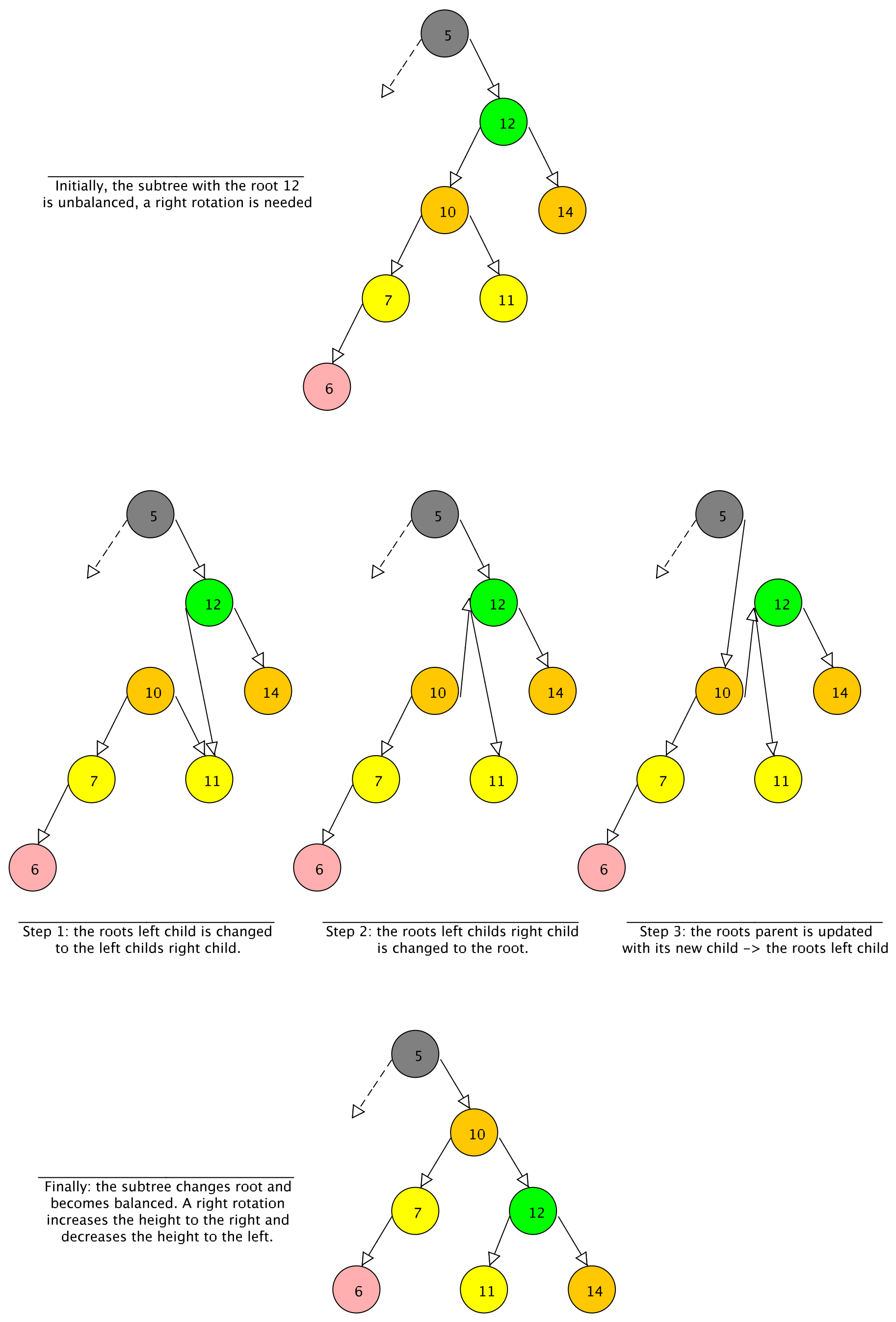
2.1.2 Deletion

Deleting a value from an AVL tree consists of two major steps: removal and re-balancing. When removing a node, there are two possible scenarios: either the node is a leaf or not. If it is a leaf, simply remove it and update the nodes parent. If the node is not a leaf, it should be replaced with either the largest node in its left subtree, or with the smallest node in its right subtree. Since the tree is balanced before the deletion, the replacement cannot have more than one child. If the replacement is a leaf, simply remove it. If the replacement has a child, update the replacements parent with its child as the parents new child.

The node has now been removed, either by replacement or by simple removal. Now a re-balancing has to be performed. This is done by retracing all the way to the root, starting with the parent of the removed node, or in the case of a replacement, with the parent of the replacement node. At each node the balance factor is updated. If the balance factor becomes 1 or -1, this indicates that the removal of the node did not change the height of the subtree. In this case the retracing can stop without going all the way to the root. If the balance factor becomes 0, the deletion did change the height of the subtree and retracing has to continue. The last case is that the balance factor becomes 2 or -2. This requires the same rotations as when inserting values. When the rotations are finished the balance factor is rechecked, if it has become 0, the retracing has to continue, otherwise the entire deletion is completed.

2.2 Special tree operations

Before implementing an AVL tree some basic tree operations are needed. These are called rotations and makes it possible to change the balance in a tree. There are two kinds of rotations: a left rotation and a right rotation. These operations mirror each other and work by increasing the height of one side of the tree by one, while decreasing that of the other by one. Figures 2 and 3 illustrate how these rotations are made.



Initially, the subtree with the root 12 is unbalanced, a right rotation is needed

Step 1: the roots left child is changed to the left childs right child.

Step 2: the roots left childs right child is changed to the root.

Step 3: the roots parent is updated with its new child -> the roots left child

Finally: the subtree changes root and becomes balanced. A right rotation increases the height to the right and decreases the height to the left.

Figure 2: A right rotation, step by step

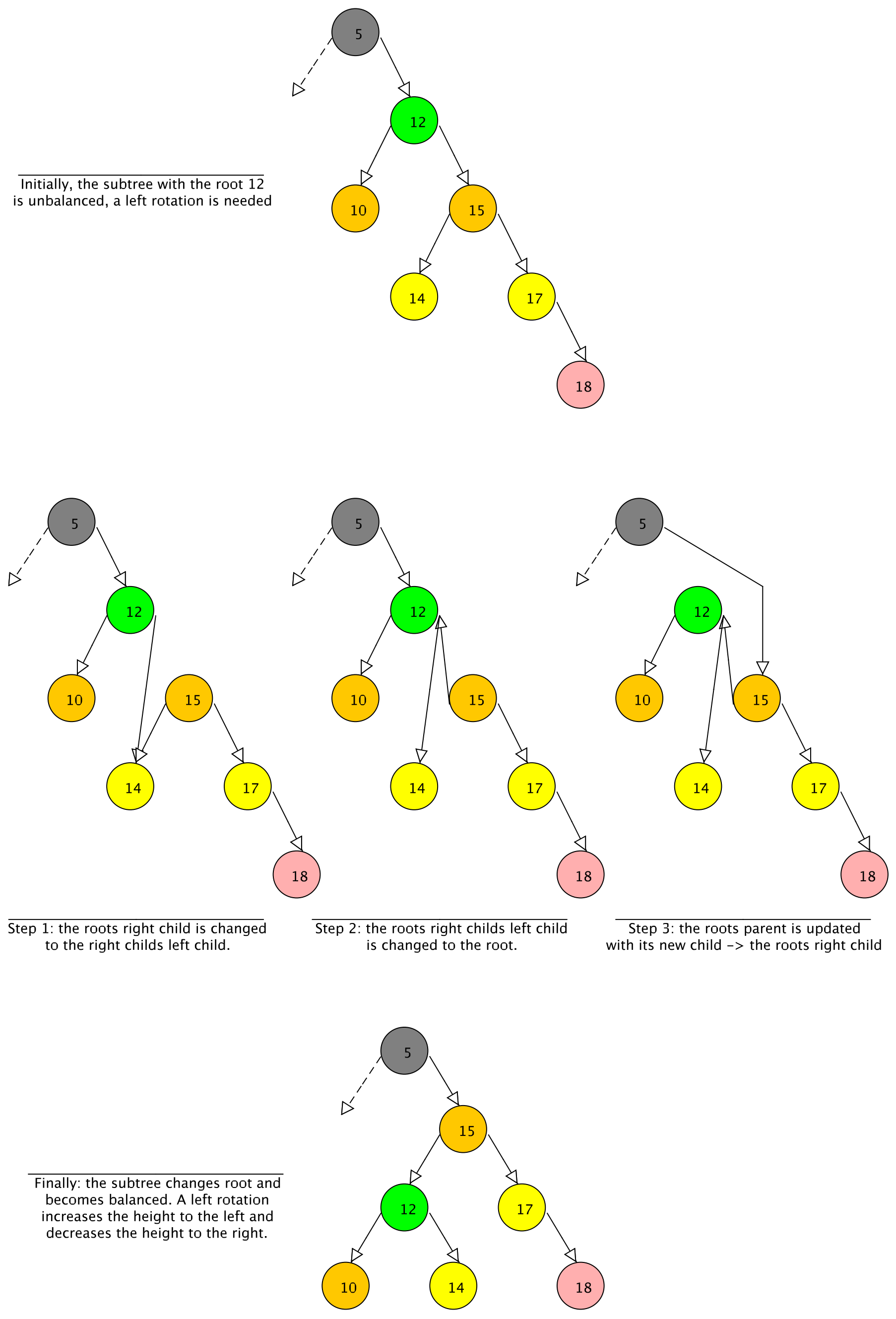

Initially, the subtree with the root 12
is unbalanced, a left rotation is needed
5
12
10
15
14
17
18
Step 1: the roots right child is changed
to the right childs left child.
Step 2: the roots right childs left child
is changed to the root.
Step 3: the roots parent is updated
with its new child -> the roots right child
Finally: the subtree changes root and
becomes balanced. A left rotation
increases the height to the left and
decreases the height to the right.

Figure 3: A left rotation, step by step