CMSC 420: AVL Trees

Justin Wyss-Gallifent

February 22, 2023

1 Definition

Binary search trees can become very unbalanced and so what we'd like to do is keep them balanced. First of all though, we need to rigorously defined what "balanced" means.

Definition 1.0.1. For a node v we define the *balance* of v:

$$b(v) = h(v.right) - h(v.left)$$

Definition 1.0.2. We say that a binary search tree satisfies the *AVL balance condition* and is an *AVL tree* if $b(v) \in \{-1, 0, 1\}$ for all nodes v in the tree.

2 Tree Height

Theorem 2.0.1. An AVL tree with n nodes has height $\mathcal{O}(\lg n)$.

Proof. Let $N(h)$ be the minimum number of nodes in an AVL tree of height h. Observe that $N(0) = 1$ as a tree has height 0 iff it is a single node. In an AVL tree the heights of the subtrees could be equal but we are trying to minimize the number of nodes which means keeping the heights as small as necessary. This means we should assume that one subtree is shorter than the other by 1, meaning one has height $h - 1$ and the other $h - 2$.

Without loss of generality then assume that the left subtree is higher than the right subtree, meaning the left subtree has height $h - 1$ and the right subtree has height $h - 2$. It follows that:

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

And also that:

$$N(h - 1) > N(h - 2)$$

From these we get the following:

$$N(h) = 1 + N(h - 1) + N(h - 2) > 1 + 2N(h - 2) > 2N(h - 2)$$

From here a pattern emerges.

- When h is even we get:

  $$N(h) > 2N(h - 2) > 2^2 N(h - 4) > ... > 2^{h/2} N(0) = 2^{h/2}$$

  Taking the lg of both sides yields:

$$\lg N(h) > \frac{h}{2}$$

And so:

$$h < 2 \lg N(h)$$

Since $N(h)$ is the minimum number of nodes in an AVL tree of height h then for any AVL tree of height h if there are n nodes then $N(h) \leq n$ and so:

$$h < 2 \lg n$$

- When h is odd the argument is similar but a bit more awkward. We'll omit it for now.

It follows from these two cases that $h = \mathcal{O}(\lg n)$.

$\mathcal{QED}$

Note 2.0.1. It's possible to get a better (meaning smaller) constant than $C = 2$. If you are interested please check David Mount's notes. I am sticking with this proof because it is simpler.

We can now see that if we can keep the tree balanced then any operation that is constrained by the height will be faster than for a binary search tree.

3 Search

3.1 Algorithm

At this point we reiterate that an AVL tree is a binary search tree at heart, meaning search is exactly the same.

3.2 Worst-Case Time Complexity

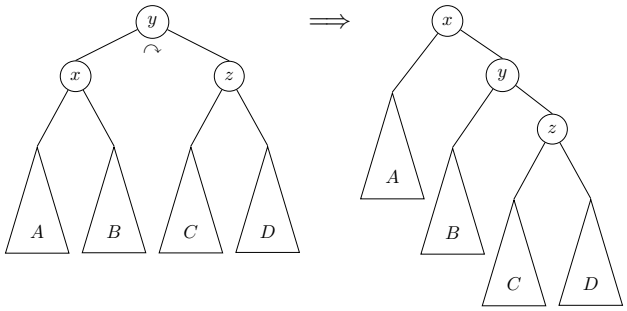
Since the height of the tree satisfies $\mathcal{O}(\lg n)$ it follows that search is also $\mathcal{O}(\lg n)$.

4 Keeping Balance with Rotations

The tricky thing of course is preserving the balance condition when we insert and delete. To do this we use four operations. In the following diagrams the triangles represent arbitrary sized subtrees and the root node shown is typically just some node, it being the root of its own subtree.

4.1 Right Rotation

Right rotation of a node works as follows:



Observe the following very important points:

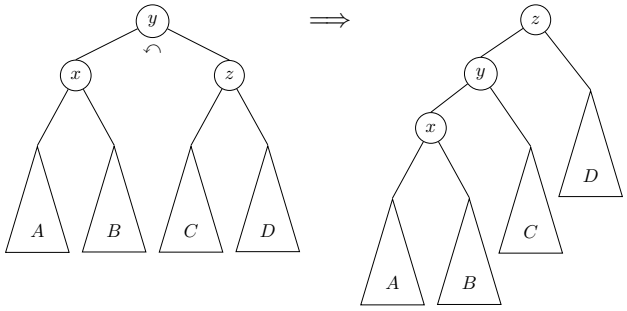
- The subtree still satisfies the BST condition:

$$A < x < B < y < C < z < D$$

- The left-left subtree moves up one level.
- The left-right subtree stays at the same level.
- The right subtree moves down one level.

4.2 Left Rotation

Left rotation of a node works as follows:



Observe the following very important points:

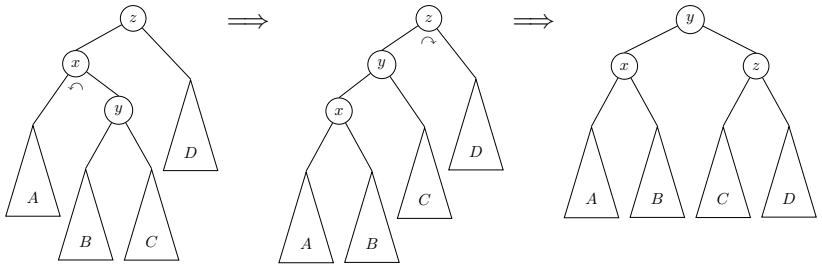
- The subtree still satisfies the BST condition:

$$A < x < B < y < C < z < D$$

- The left subtree moves down one level.
- The right-left subtree stays at the same level.
- The right-right subtree moves up one level.

4.3 Left-Right Rotation

A left rotation on a left child followed by a right rotation on the parent.



Observe the following very important points:

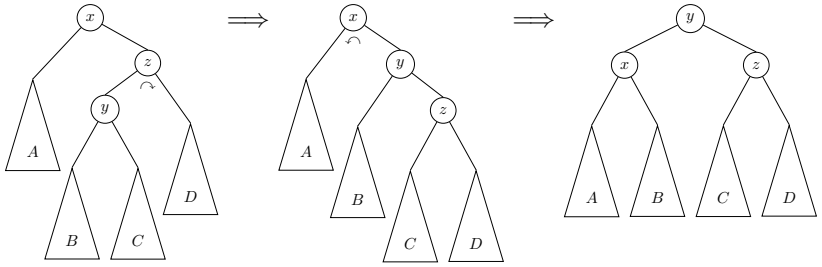
- The subtree still satisfies the BST condition:

$$A < x < B < y < C < z < D$$

- The left-left subtree stays at the same level.
- The left-right subtree gets broken up and both of the children move up one level.
- The right subtree moves down one level.

4.4 Right-Left Rotation

A right rotation on a right child followed by a left rotation on the parent.



Observe the following very important points:

- The subtree still satisfies the BST condition:

$$A < x < B < y < C < z < D$$

- The left subtree moves down one level.
- The right-left subtree gets broken up and both of the children move up one level.
- The right-right subtree stays at the same level.

5 Insertion

5.1 Algorithm

For insertion, we first insert as with a BST. The only nodes which may now violate the balance condition are nodes along the path from the newly inserted (leaf) node up to the root, so we must traverse from the newly inserted leaf up to the root and check.

It might be tempting to believe that we have to check every node but it turns out that if we encounter one unbalanced node and fix it, we are actually done.

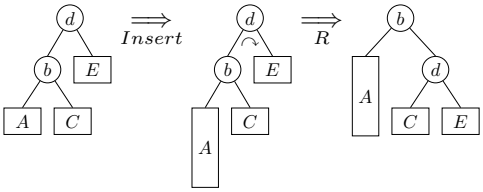
It should be noted before proceeding that we insert a new leaf node. The parent of that new leaf node is simply gaining a child. Perhaps it had another child, perhaps not, but either way it will not violate the balance condition so really we only need to start checking the grandparent of the newly inserted node.

There are two cases each with two subcases:

1. Left-heavy: The unbalance arises in the left subtree.

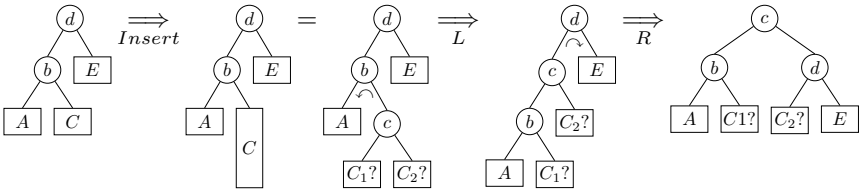
   (a) Left-left heavy.

   In the case where the unbalance is in the left subtree of the left subtree a right rotation will fix the balance condition. This is shown by the following diagram. In the leftmost tree below, subtrees A and C extend exactly one level deeper than subtree E.



   (b) Left-right heavy: In the case where the unbalance is in the right subtree of the left subtree a left-right rotation will fix the balance condition. This is shown by the following diagram. In the leftmost tree below, subtrees A and C extend exactly one level deeper than subtree E.

   Note that if c is the root node for the subtree C then when we insert into subtree C we actually insert into either the left or right subtree of c. We're not sure which, but one of those:



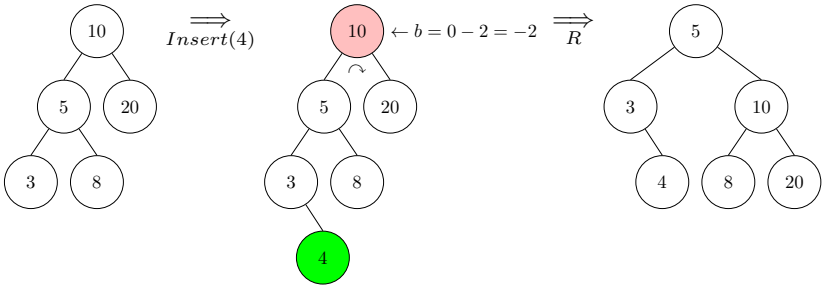
2. Right-heavy: The unbalance arises in the right subtree.

(a) Right-right heavy: In the case where the unbalance is in the right subtree of the right subtree a left rotation will fix the balance condition. The argument is symmetric to above.

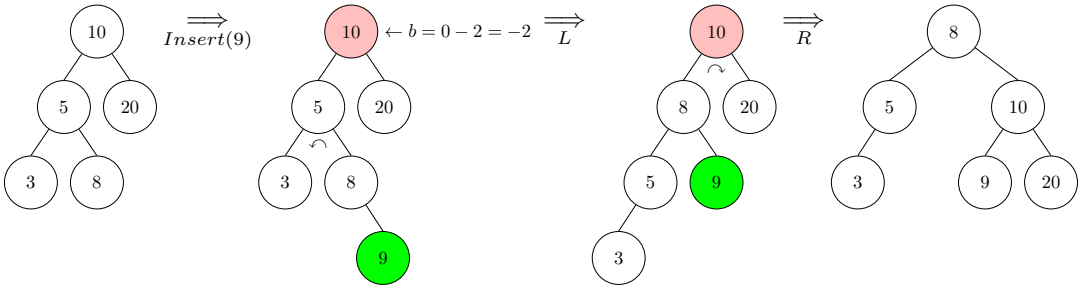
(b) Right-left heavy: In the case where the unbalance is in the left subtree of the right subtree a right-left rotation will fix the balance condition. The argument is symmetric to above.

Note 5.1.1. Notice now that in all four of these cases the balance of the root node is 0. Additionally the height of the resulting subtree is exactly the same as it was before the insertion. Effectively this means that all of the nodes further up will be balanced and we do not need to fix them.

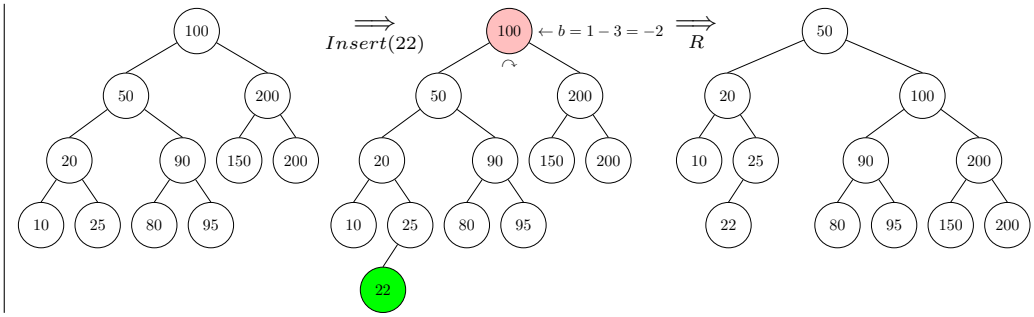
Example 5.1. An example of insertion resulting in a left-left heavy tree, fixed by a right rotation:



Example 5.2. An example of insertion resulting in a left-right heavy tree, fixed by a left-right rotation:



Example 5.3. Another example of insertion resulting in a left-left heavy tree, fixed by a right rotation:



5.2 Worst-Case Time Complexity

The actual rebalancing operation is $\mathcal{O}(1)$ and only happens once. However we may need to do $\mathcal{O}(\lg n)$ node inspections (due to the height) in order to see if there is an unbalanced node, thus insertion is $\mathcal{O}(\lg n)$.

6 Deletion

6.1 Algorithm

For deletion we first delete as usual. Recall that with deletion there is a replacement process which ends with either a leaf node being deleted or a non-leaf node with one child being deleted and the single subtree being promoted.

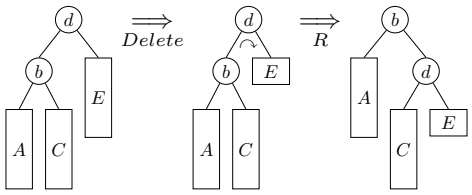
Replacement of nodes by nodes further down the tree does not affect heights or balances. Those are only affected once we actually delete or delete and promote.

If it is a leaf node we finally delete then it is possible that its parent is unbalanced so we need to start checking with that parent.

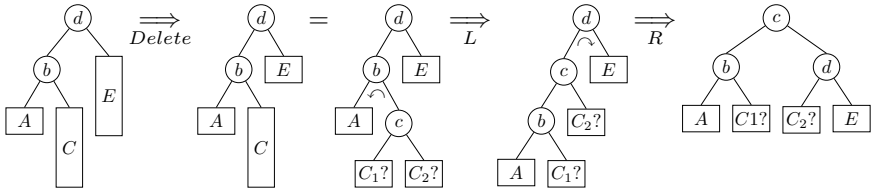
If it is not a leaf node we finally delete then all the promoted nodes are safe since their balance factors do not change. We do however need to check the parent of the deleted node.

There are two cases each with two subcases:

1. Left-heavy: The unbalance arises in the left subtree.
   (a) Left-left heavy (and maybe left-right heavy). In the case where a deletion in the right subtree causes an unbalance in the left subtree of the left subtree a right rotation will fix the balance condition. This is shown by the following diagram. In the leftmost tree below, subtree A extends exactly one level deeper than subtree E while subtree C extends either to the same level as A or one less.



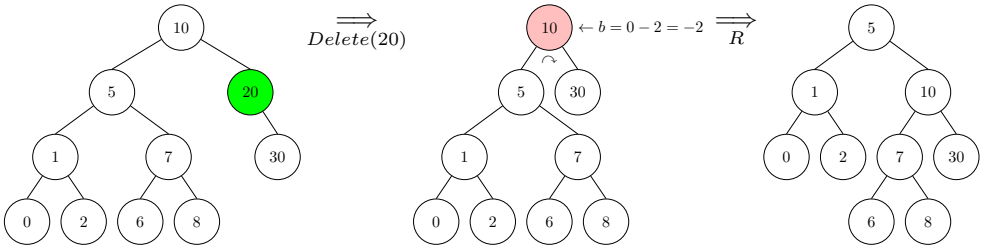
(b) Left-right heavy (but not left-left heavy). In the case where a deletion in the right subtree causes an unbalance in the right subtree of the left subtree (and not in the left subtree of the left subtree) a left-right rotation will fix the balance condition. This is shown by the following diagram for which all but the leftmost tree is exactly the same as the left-right heavy insert case:



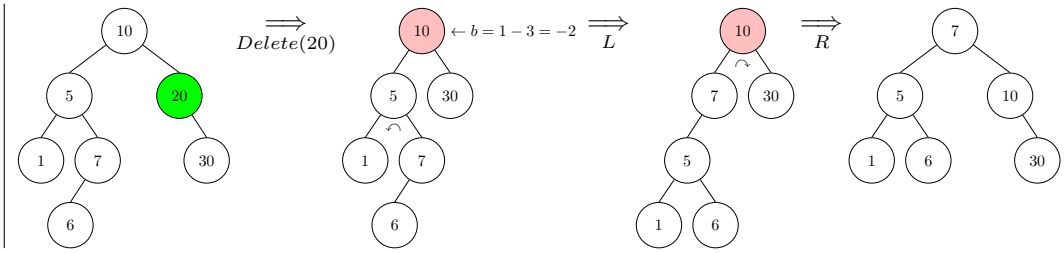
2. Right-heavy: The unbalance arises in the right subtree.
   (a) Right-right heavy: The argument is symmetric to above.
   (b) Right-left heavy: The argument is symmetric to above.

Notice now that in all four of these cases the balance of the root node is in $\{-1, 0, 1\}$. Unlike insertion the height of the resulting subtree may have changed from what it was before the insertion. Effectively this means that we will need to move up the tree to potentially fix the balance of those nodes.

Example 6.1. An example of deletion resulting in a left-left heavy tree fixed by a right rotation:



Example 6.2. An example of deletion resulting in a left-right heavy tree fixed by a left-right rotation:



6.2 Worst-Case Time Complexity

We saw that the BST deletion operation is constrained by height, so that is $\mathcal{O}(\lg n)$ in this case. The rebalancing operation is $\mathcal{O}(1)$ but we have to inspect all the way up the tree from the deletion point, and multiple rebalances may be needed. Thus this is $\mathcal{O}(\lg n)$.

7 Height Calculations

During this discussion we have constantly referred both to heights of subtrees and to node balances (which are calculated from heights of subtrees).

We may wonder how we will know the heights. The answer is that we can store them in the nodes and update them as necessary. Effectively each node can contain the height of that node and then when insertions and deletions are performed we update the heights accordingly.