# CMSC 420: AA Trees

Justin Wyss-Gallifent

January 6, 2023

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Definitions</td>
<td>2</td>
</tr>
<tr>
<td>2.1 Red-Black Tree</td>
<td>2</td>
</tr>
<tr>
<td>2.2 Levels and Tree Height</td>
<td>3</td>
</tr>
<tr>
<td>2.3 AA Tree</td>
<td>4</td>
</tr>
<tr>
<td>Search</td>
<td>4</td>
</tr>
<tr>
<td>Balancing Operations</td>
<td>4</td>
</tr>
<tr>
<td>4.1 Skew (Right Rotate)</td>
<td>4</td>
</tr>
<tr>
<td>4.2 Split (Left Rotate)</td>
<td>5</td>
</tr>
<tr>
<td>4.3 UpdateLevel</td>
<td>5</td>
</tr>
<tr>
<td>Insertion</td>
<td>5</td>
</tr>
<tr>
<td>5.1 Algorithm</td>
<td>5</td>
</tr>
<tr>
<td>5.2 Time Complexity</td>
<td>7</td>
</tr>
<tr>
<td>Deletion</td>
<td>8</td>
</tr>
<tr>
<td>6.1 Algorithm</td>
<td>8</td>
</tr>
<tr>
<td>6.2 Time Complexity</td>
<td>12</td>
</tr>
</tbody>
</table>
1 Introduction

There are several issues with 2-3 trees, not that the adjustment process associated to insertion and (especially) deletion are fairly complicated and involves many cases but also that we have to store two different types of nodes, and occasionally convert between them, which makes the tree management somewhat arduous.

Since binary trees are so much simpler we might ask if it's possible to encode a 2-3 tree as some sort of augmented binary tree and adjust insertion and deletion accordingly.

The short answer is yes.

Imagine taking a 2-3 tree with subtrees $A$, $B$, and $C$ and for each 3-node we replace it by two 2-nodes as shown here:

We have made the right-hand node red not just to remind us that it was originally on the same level as its parent but (as we’ll see) to ensure that our tree still “looks perfect”.

A more comprehensive example can be seen if we did this with an actual tree:

Example 1.1. For example:

2 Definitions

2.1 Red-Black Tree

Definition 2.1.1. We define a red-black tree to be a binary search tree which satisfies the following conditions:
(a) Each node is either red or black.
(b) The root is black.
(c) If a node is red, both its children are black.
(d) All null pointers are treated as if they point to black nodes.
(e) Every path from a given node to any of its null descendents contains the same number of black nodes.

2.2 Levels and Tree Height

Condition (e) from the definition makes it clear that red nodes are “halfway down” in the sense that it is the black nodes which determine the levels of the tree and which determine the fact that the tree is “perfect”.

More rigorously we define the level of each node:

**Definition 2.2.1.** The level of a node is defined as follows:

- The lowest black nodes are said to be at level 1.
- As we move up the tree the levels of black nodes increase.
- The level of a red node is equal to that of its parent.
- The lowest nodes do not of course have children but for the sake of argument we’ll say that their NULL children have level 0. This will make some discussions easier.

In truth the third point actually suggests that we don’t really need the nodes to be red, rather we simply define a node to be red iff it is the same level as its parent. We will however keep the nodes red just for a visual aid.

**Example 2.1.** Here is the above example with levels indicated:

![Image of a red-black tree with levels indicated](image)

**Definition 2.2.2.** The height of a tree is then the difference between the root node level and the leaf node level.

Observe that because of the red nodes if we search through a red-black tree as if it were a binary search tree the traversal is not constrained by the height of the tree as defined but is constrained by twice that height, so $h = \Theta(\lg n)$ anyway.
2.3 AA Tree

It turns out that red-black trees are not equivalent to 2-3 trees but rather to 2-3-4 trees (whose definition ought to be clear). This is because, for example, in a red-black tree a node may have a red left child but this doesn’t arise as equivalent to anything in a 2-3 tree.

In order to establish equivalence we add one more condition:

**Definition 2.3.1.** We define an AA Tree to be a red-black tree such that:

(f) Each red node can arise only as the right child of a black node.

3 Search

An AA tree is a binary search tree and so search is \( \Theta \) of the height, hence \( \Theta(\lg n) \).

4 Balancing Operations

The major convenience of using AA trees in place of 2-3 trees is the relative simplicity in the balancing operations.

It turns out that only three balancing operations are required, called skew, split, and updateLevel.

4.1 Skew (Right Rotate)

Here is the skew operation. It's actually just a right rotation and so it should be familiar to you! You may look at the tree on the left and notice it's not an AA tree. This is correct, but as we'll see it can arise as the result of an insert or delete and so the skew operation returns it to AA tree status.

Importantly note that the node \( y \) could be red or black and neither of \( x \) nor \( y \)’s colors/levels change.
4.2 Split (Left Rotate)

Here is the split operation. It’s actually just a left rotation and so it should be familiar to you! Again you may look at the tree on the left and notice it’s not an AA tree, or even a red-black tree. This is correct, but as we’ll see it can arise as the result of an insert or delete and so the split operation returns it to AA tree status.

Importantly note that nodes $x, y, z$ were all at the same level and how $x$ and $z$ have remained at that level but $y$ has been pushed up to become the same level as its parent.

4.3 UpdateLevel

Unlike skew and split, UpdateLevel does not restructure the tree, rather it simply makes some changes to the levels.

For a specific node, UpdateLevel checks to see if the node is more than one level higher than its lowest child. If it is too high, it lowers it accordingly. In addition if the node had a right child then that right child is lowered as well.

For purposes of counting height a null node is considered to be black and at level 0.

This will not be used for insert but only for delete.

5 Insertion

5.1 Algorithm

Consider the insertion of a new key. We begin by inserting as with a standard BST with the caveat that we assign the level of the new node to be the same level as its parent. This is essentially the same as saying it is red.
At this point there is a potential chain of corrections that need to be made but luckily this chain consists of only two problems:

- If we have a red node as the left child of a node then skew.
- If we have a red node as the right child of a red node then split. The split operation is the one which sends a node up to be the same level as its parent and the chain may continue.
Example 5.1. Let’s trace through an example. Consider the following:

5.2 Time Complexity

Each of the skew and split operations is \( \Theta(1) \) and we may need \( \Theta(\lg n) \) of them, yielding a time complexity of \( \Theta(\lg n) \).
6 Deletion

6.1 Algorithm

As usual deletion is more complicated than insertion. We begin as with usual BST deletion and since every node (except the leaves) has a left child we know it’s a node in level 1 that will eventually be removed.

If that node is red, we just delete it and we are done. If that node is black with a red child then we remove the node and promote the red child to black. If neither of these are true then we have work to do. and we begin a restructuring process which goes up the tree.

When we visit a node \( p \) we first pull down its level if needed using UpdateLevel. Note that if it has a red right child (right child at the same level) then that child gets pulled down too.

Once that node is pulled down we may have red left children at the same level as their parents a red node may have a red right child. In order to fix this we do two things:

1. Follow the right path from \( p \) and if a node has a red left child, do skew. This will fix the first issue.
2. Follow the right path from \( p \) and if a node is red and has a red right child, do split. This will fix the second issue.

This process gets repeated for every node which is pulled down!

Example 6.1. Let’s trace through a simple example first. Consider the following:

Let’s delete the key 17:
We first note that the parent node with key 18 is on level 2 but since it has an empty left child (which is taken to be level 0) it should be on level 1 so we move it down. The 19 is still its child so the 19 becomes red:

Now 18 has been dealt with so we move up to 20 which is at level 3 but should be at level 2. When we do this the 30 now turns red:

Now we are done. In this case it was only UpdateLevel that we needed.

**Example 6.2.** Let’s trace through a more complicated example. I stole this from Arne Andersson’s original paper. Consider the following:
Let’s delete the key 1.

We first note that the parent node with key 2 is on level 2 but since it has an empty left child (which is taken to be level 0) so we move it down. The 3 is still its child so the 3 becomes red:

Now 2 has been dealt with so we move up to 4 which is at level 3 but should be at level 2. When we do this the 6 and 12 now turn red:

Now 4 is fine as far as its own level but that entire level is a mess. We begin the skew operation(s). We move down to 10 which has a red left child so we skew:
Note that 10 has been replaced by 6 so we move down to the right child which is 10 again and has a red left child so we skew:

Note that 10 has been replaced by 8 so we move down to the right child which is 10 again but now it has no red left child so the skewing is all done. Here’s the picture so far again for reference:

Next we go back up to 4 and observe that it has a red right child with a red right child so we split. The 6 moves up to the top level:

The 4 has been replaced by 6 so we go to 6’s right child which is 8, and this has a red right child with a red right child so we split. The 10 moves up a level:

Now we are done.
6.2 Time Complexity

Each of the associated operations is $\Theta(1)$ and we may need $\Theta(\lg n)$ of them, yielding a time complexity of $\Theta(\lg n)$. 