CMSC 420: AA Trees

Justin Wyss-Gallifent

March 9, 2023

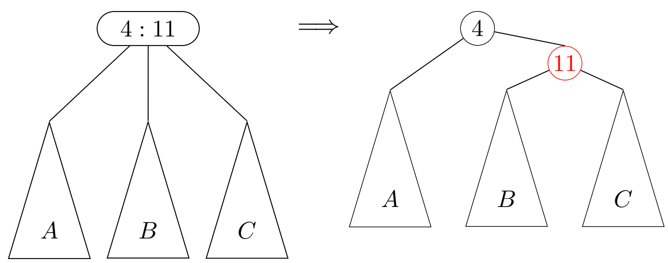
1 Introduction

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.

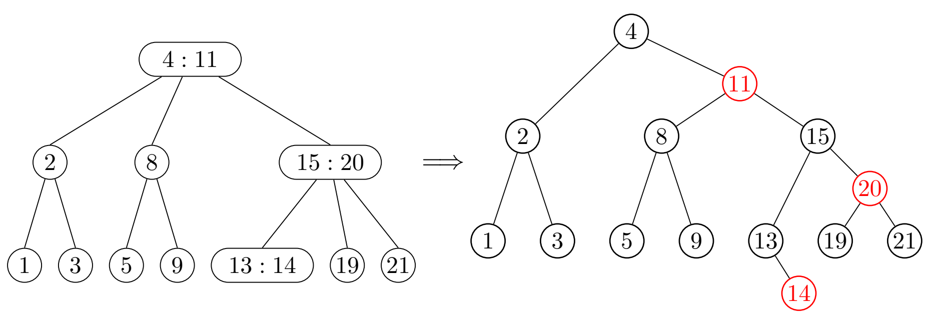
The short answer is yes.

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".

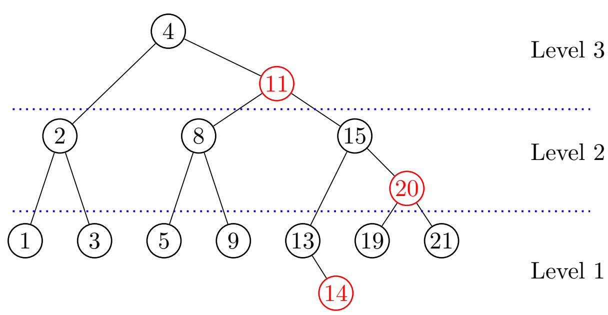
More rigorously we define the level of each node:

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:



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 $h = \mathcal{O}(\lg n)$ 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 $\mathcal{O}$ of the height, hence $\mathcal{O}(\lg n)$.

4 Balancing Operations

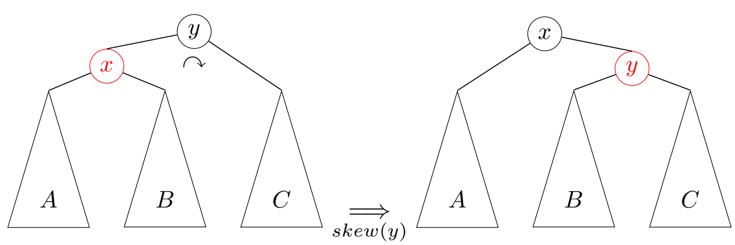
The major convenience of using AA trees in place of 2-3 trees is the relative simplicity in the balancing operations.

It turns out that only three balancing operations are required. These are called skew, split, and update-level.

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!

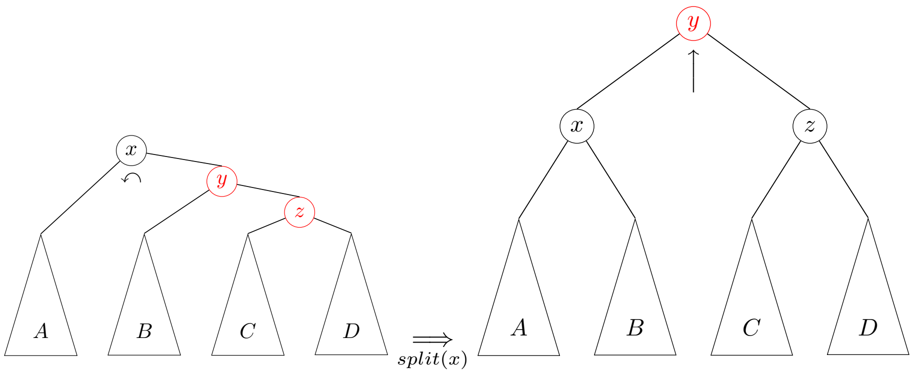
Consider the following diagram. In the left tree y could be red or black but x is definitely red. The skew operation fixes the existence of a red left child. In the right tree the levels of x and y are unchanged and consequently y is certainly red but x could be either red or black, depending on its parent.



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!

Consider the following diagram. In the left tree x is definitely black and y and z are definitely red. The split operation fixes the existence of a red child of a red child. In the right tree y has been promoted to the same level as its parent, so it is red, and the levels of x and z are unchanged, and they are black.



4.3 Update-Level

Unlike skew and split, update-level does not restructure the tree, rather it simply makes some changes to the levels. It will only be used for delete.

For a specific node, update-level 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.

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.

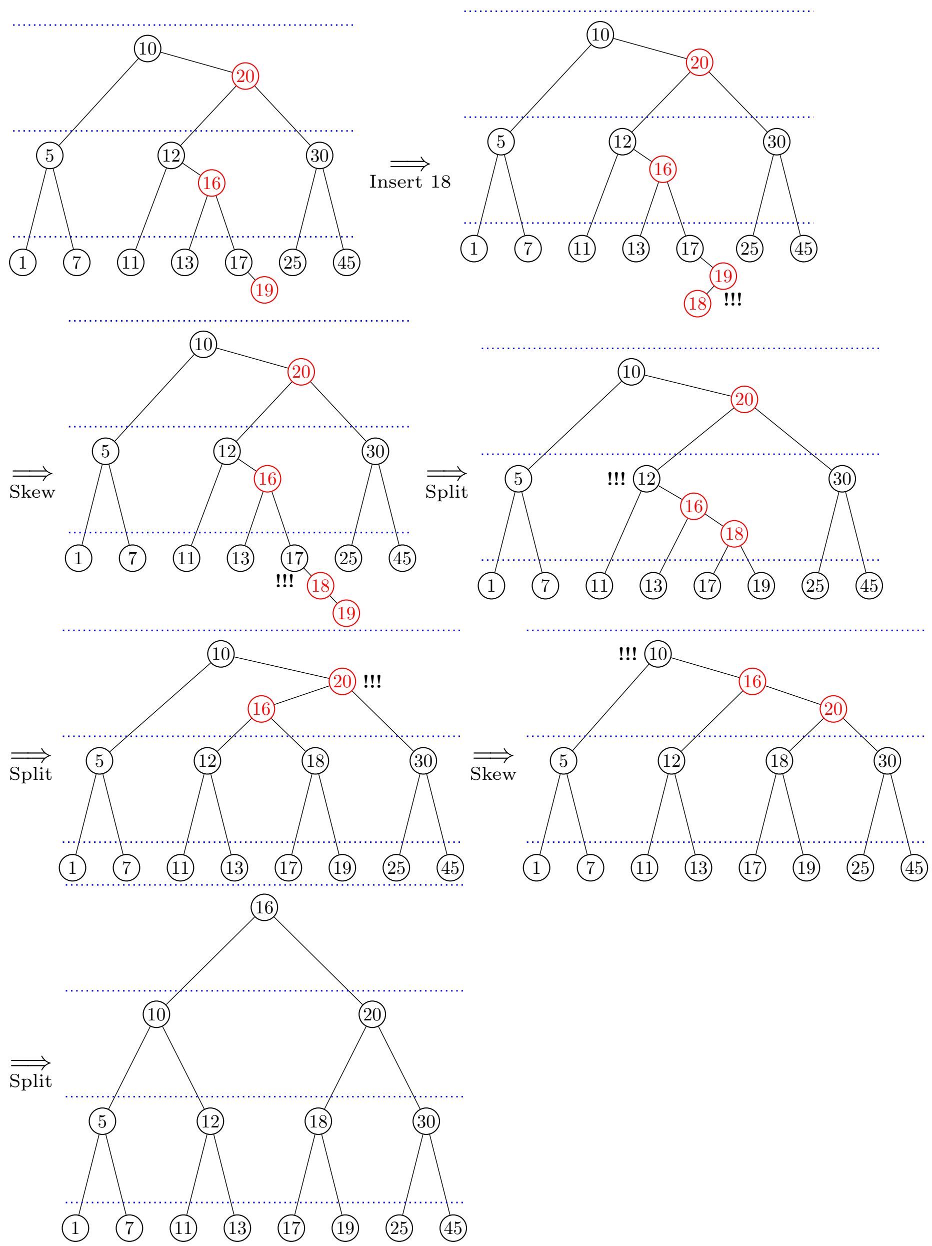
At this point we must fix any issues. Starting at the inserted node:

- If we have a red node as the left child of a node then skew.
- If we we have a red node as the right child of a red node then split.

If either of these occur we must then check the parent of the root of the subtree we just skewed or split and this may propagate all the way to the root.

Example 5.1. Let's trace through an example.

Consider the following:



5.2 Time Complexity

Each of the skew and split operations is $\mathcal{O}(1)$ and we may need $\mathcal{O}(\lg n)$ of them, yielding a time complexity of $\mathcal{O}(\lg n)$.

6 Deletion

6.1 Algorithm

As usual deletion is more complicated than insertion. We begin as with usual BST deletion process (finding a replacement) 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, that node is the root of a subtree. For what follows denote by p the root of that subtree

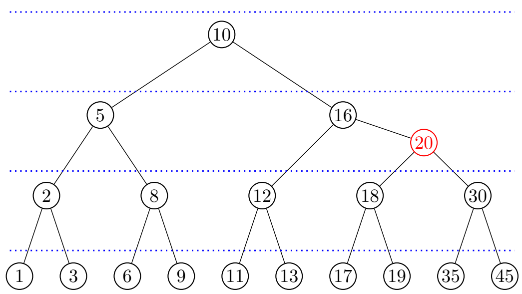
We first pull down p if needed using update-level. If we pull it down we may create some problems.

First we must make sure that there are no red left children. We can fix this with at most three calls to skew: the first called on p, the second called on $p.right$, and the third called on $p.right.right$. Each of these is only called if it fixes a red left child.

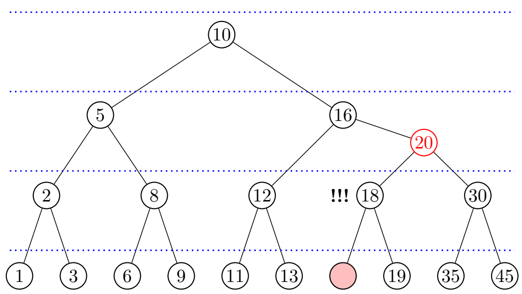
After that at worst p will be a black node with a chain of four red right descendents. We can fix this with at most two calls to split: the first called on p and the second called on $p.right$. Each of these is only called if it fixes a red right child of a red right child.

In total this is $\mathcal{O}(1)$. This process gets repeated for every node which is pulled down!

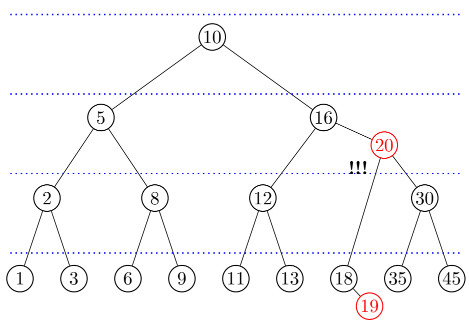
Example 6.1. Let's trace through a simple example first. Consider the following:



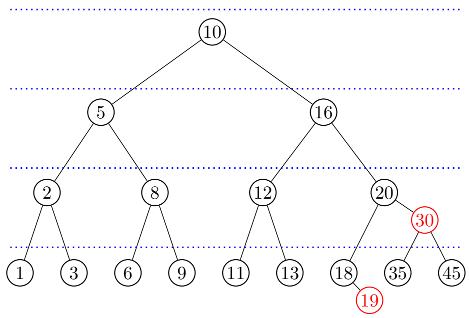
Let's delete the key 17:



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:

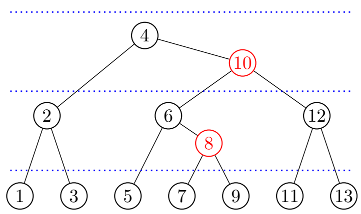


Now 18 has been dealt with so we move up to 20 which is at level 3 but should be at level 2. When we do this the 30 now turns red:

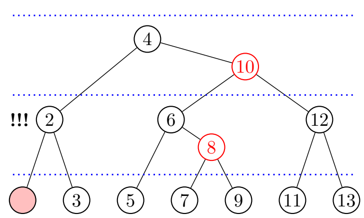


Now we are done. In this case it was only update-level that we needed.

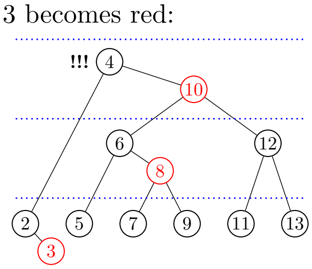
Example 6.2. Let's trace through a more complicated example. I stole this from Arne Andersson's original paper. Consider the following:



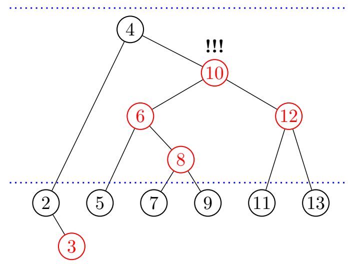
Let's delete the key 1.



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:

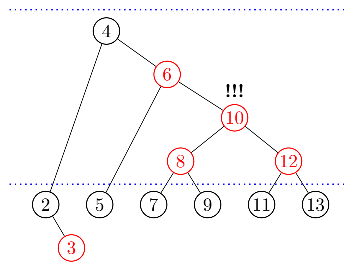


Now 2 has been dealt with so we move up to 4 which is at level 3 but should be at level 2. When we do this the 6 and 12 now turn red:

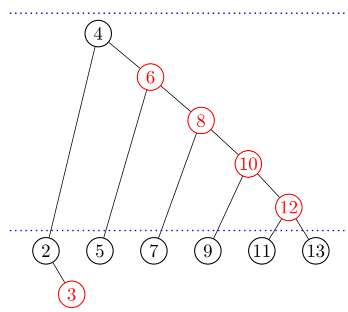


The skewing begins!

The root is 4. We don't need to skew 4 as it has no red left child. We do need to skew $4.right = 10$ since it has a red left child:



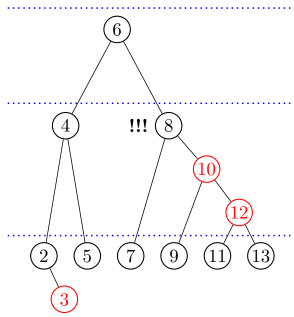
Now we need to skew $4.right.right = 10$ since it has a red left child:



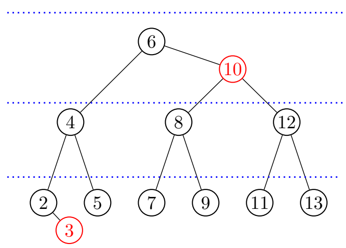
Now the skewing is done.

The splitting begins!

The root is 4. We need to split 4 since it has a red right child with a red right child:



Now the root is 6. We need to split $6.right = 8$ since it has a red right child with a red right child:



Now we are done.

6.2 Time Complexity

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