CMSC 420: 2-3 Trees

Justin Wyss-Gallifent

March 5, 2023

1 Introduction

For a 2-3 tree we generalize binary search trees a little by allowing the existence of both 2-nodes and 3-nodes, meaning nodes with 2 or 3 children (or 0 for the non-leaves) and 1 or 2 keys respectively. Note that in such a tree the number of nodes and the number of keys may be different.

By doing this we can ensure the tree is perfect, meaning all levels are full, and that the height is logarithmic both on the number of nodes and the number of keys.

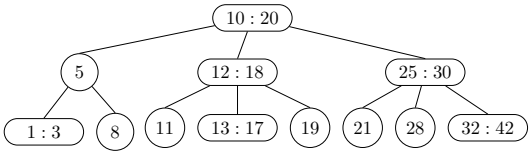
2 Definition

More rigorously we define:

Definition 2.0.1. A 2-3 tree of height $h \geq -1$ is defined as follows:

- It is empty iff $h = -1$, otherwise:
- A 2-node b with left subtree A and right subtree C satisfies $A < b < C$.
- A 3-node $b : d$ with left subtree A and middle subtree C and right subtree E satisfies $A < b < C < d < E$.
- If the root is a 2-node then the two subtrees are 2-3 trees of height $h - 1$.
- If the root is a 3-node then the three subtrees are 2-3 trees of height $h - 1$.

Example 2.1. Here is an example:



3 Tree Height and Key and Node Counts

Suppose a 2-3 tree has height h, meaning $h + 1$ levels indexed 0 through h, n nodes, and k keys.

- The sparsest 2-3 tree with height h will have all 2-nodes and hence the number of nodes will be:

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

  Since this is the sparsest it follows that for any 2-3 tree with height h we have:

$$n \geq 2^{h+1} - 1$$

And since h is an integer we have:

$$h \leq \lfloor \lg(n+1) - 1 \rfloor$$

In addition since $k = n$ in this case we get:

$$k \geq 2^{h+1} - 1$$

And:

$$h \leq \lfloor \lg(k+1) - 1 \rfloor$$

- The densest 2-3 tree with height h will have all 3-nodes and hence the number of nodes will be:

$$n = \sum_{i=0}^{h} 3^i = \frac{3^{h+1} - 1}{3 - 1} = \frac{1}{2}\left(3^{h+1} - 1\right)$$

Since this is the densest it follows that for any 2-3 tree with height h we have:

$$n \leq \frac{1}{2}\left(3^{h+1} - 1\right)$$

And since h is an integer we have:

$$h \geq \lceil \log_3(2n+1) - 1 \rceil$$

In addition since $k = 2n$ in this case we get:

$$k \leq \left(3^{h+1} - 1\right)$$

And:

$$h \geq \lceil \log_3(k+1) - 1 \rceil$$

From the above points we observe that:

$$\lceil \log_3(2n+1) - 1 \rceil \leq h \leq \lfloor \lg(n+1) - 1 \rfloor$$

And:

$$\lceil \log_3(k+1) - 1 \rceil \leq h \leq \lfloor \lg(k+1) - 1 \rfloor$$

From whence we see that $h = \Theta(\lg n) = \Theta(\lg k)$.

Note 3.0.1. The inequalities above are restrictions on the height for any given k, they are not suggesting that all of these heights will work. The fact that some heights will work at all will be borne out of the process of insertion.

Example 3.1. If a 2-3 tree is going to contain 54321 keys then the only possible heights are:

$$9 = \lceil \log_3(54321+1) - 1 \rceil \leq h \leq \lfloor \lg(54321+1) - 1 \rfloor = 14$$

We may wonder if it's the case for all $k \geq 0$ that there actually values of h satisfying the bounds above. This is akin to proving that for any integer $x \geq 1$ that we have:

$$\lceil \log_3 x \rceil \leq \lfloor \log_2 x \rfloor$$

This is easy to verify for specific examples but takes some fiddling to prove in general. One approach is to first observe that both sides are 0 when $x = 1$, Then if we focus on only integers we observe that the left side increases at powers of 3 and the right side increases at powers of 2. We then show that between any two powers of 3 there is at least one power of 2, thereby verifying that the right side is always larger. You are encouraged to work out the details!

4 Search

Finding a key is essentially just a mild modification of a BST. We compare the search key with the 1 or 2 keys in each node and follow the corresponding branch.

This means that search is $\Theta(\lg k)$ in all cases.

5 Insertion

5.1 Algorithm

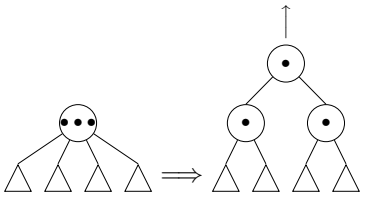
Insertion is initially like a BST, we follow the corresponding branches until we find where to put the new key.

The difference is that we may arrive at a 2-node leaf in which case we can simply insert the new key, creating a 3-node leaf, and we're done.

The issue is what happens when we arrive at a 3-node leaf. We could insert a new leaf as a child of the 3-node but what we do is slightly more elegant.

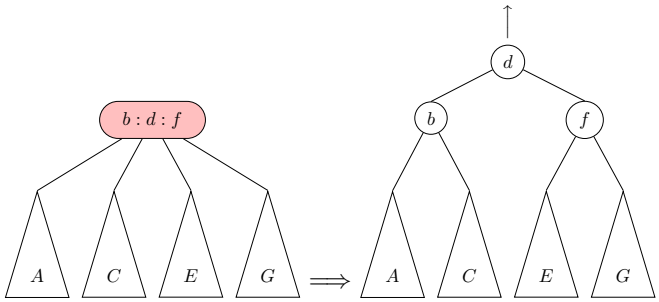
Instead we insert the new key into the 3-node, temporarily creating a 4-node, and then we engage in a process of splitting nodes and promoting keys which essentially propagates up the tree until either it is absorbed or a new root is created. The result will be a 2-3 tree.

The process is very simple and shown in the following picture.

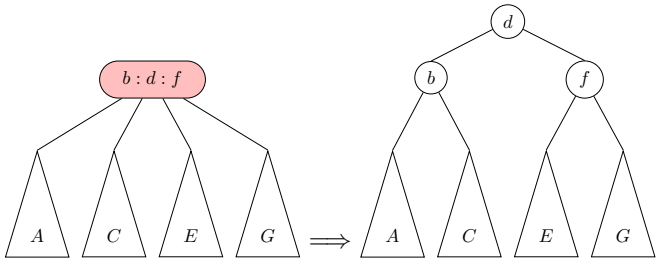


More details on what the outcome might be:

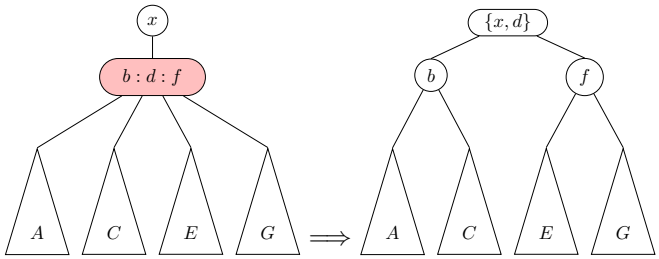
1. We split the 4-node $b : d : f$ into two 2-nodes b and f and we promote d upwards:



2. If the $b : d : f$ node had no parent then it was the root, in which case d becomes the new root and b and f are its children:

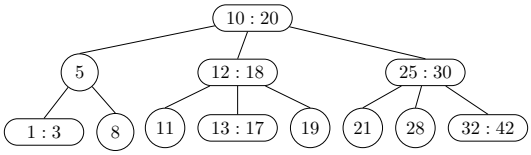


3. If that original parent is a 2-node then d is absorbed into that 2-node and it becomes a 3-node and we are done. In the following the new parent 3-node on the right will be either $x : d$ or $d : x$ as appropriate so we've just written $\{x, d\}$. Note that in the figure below the original node x has one other child on the left or right which is not shown but which still exists after the operation.

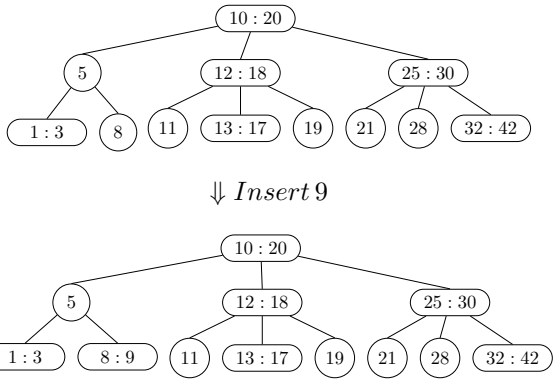


4. If that original parent is a 3-node then we pass d appropriately into that 3-node, making it a 4-node. The problem has now re-manifested one level up and so we repeat the process.

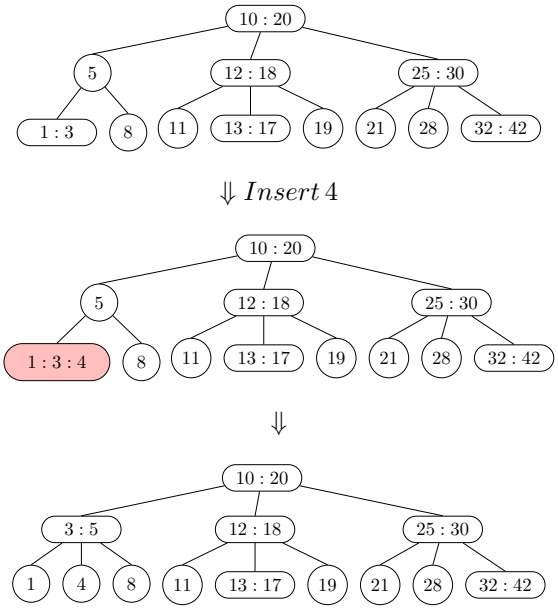
Let's work through some comprehensive examples. All three of these start with the 2-3 tree from the beginning and shown here:



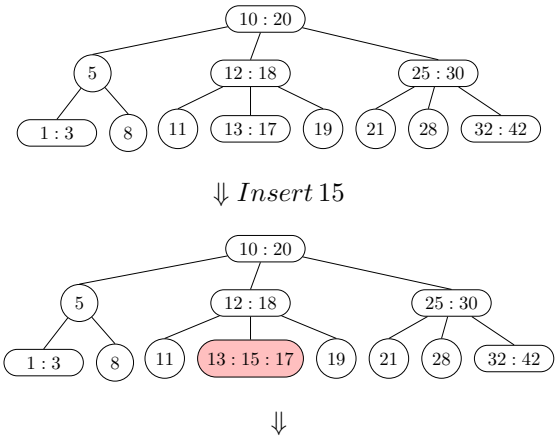
Example 5.1. If we insert 9 it just goes straight into the 2-node leaf 8 making it 8 : 9 and we're done:

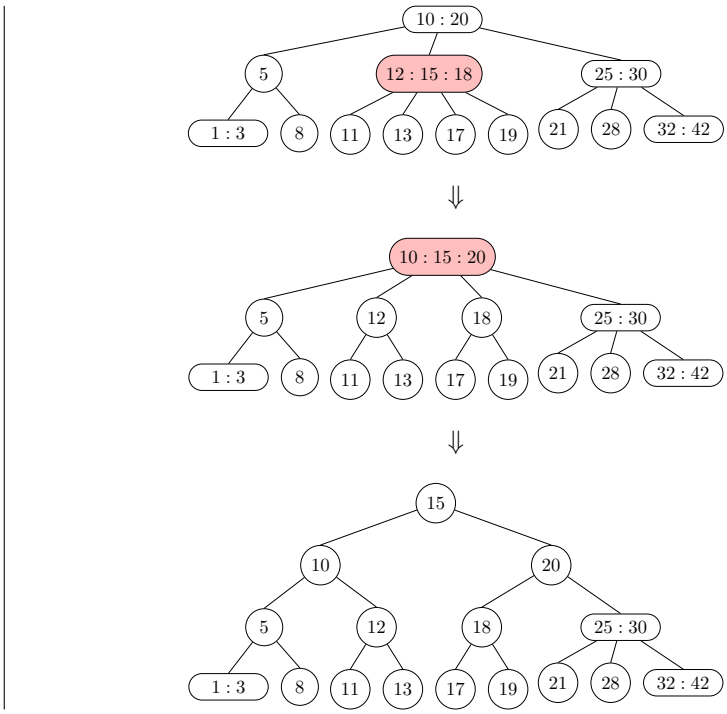


Example 5.2. If we insert 4 it just goes into the 3-node leaf 1 : 3 making it 1 : 3 : 4 which is problematic. We have to split and promote the 3 which is absorbed above it:



Example 5.3. If we insert 15 the issue propagates all the way up:





5.2 Time Complexity

The split and promote processes are each $\Theta(1)$ and since the process might propagate to the top of the tree the result is $\Theta(\lg n)$.

6 Deletion

6.1 Algorithm

We first follow a modified BST approach. We find the key we wish to delete. If it is not in a leaf node we find the inorder successor key and replace it, then recursively delete the inorder successor. Unlike standard BST deletion we never encounter a node with one child and so we never simply delete and promote.

Note 6.1.1. Consider how you might algorithmically find the inorder successor!

Consequently the result of our replacement process is that we may assume that we are deleting a key from a leaf node.

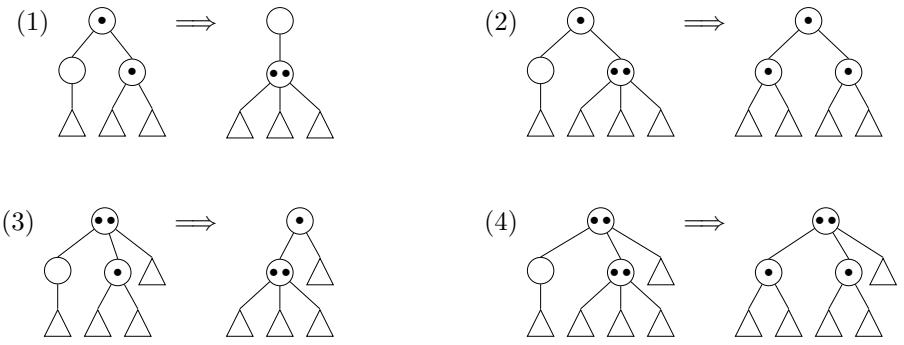
If the leaf node is a 3-node we can simply throw out the key to create a 2-node and we are done, but if the leaf node is a 2-node then we have a problem since we cannot simply remove the node.

The solution ends up being a process that can propagate up the tree. We'll cover the cases here and then see how they apply:

Let's say that a *hole* is a 1-node (with 1 child, or 0 if it's a leaf). There are four cases. Each case has variants so we present a general rule for each as well as one example. We encourage the reader to work out more examples.

In the following cases assume we are at a hole. Some of the cases overlap and there are options.

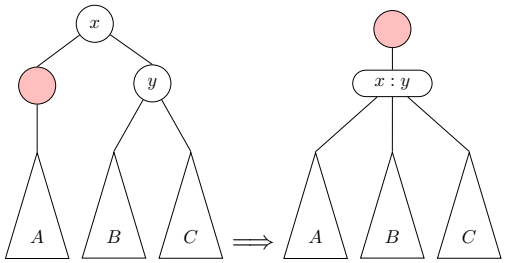
Here are the four cases as simplified diagrams, then more details are given separately:



More details:

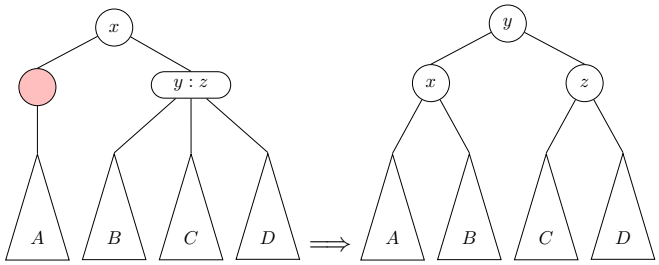
(1) If the parent is a 2-node and the sibling is a 2-node: Demote the parent to the sibling to make the sibling a 3-node and attach all three subtrees appropriately. The hole propagates up the tree. If there is no "up the tree" then we have replaced the root and are done. Important! Consider the variations on the arrangements!

Definition 6.1.1. This is called a *merge* because we are merging the 1-node with a sibling (and demoting a key from the parent). This same operation with the same name will appear in B-trees.



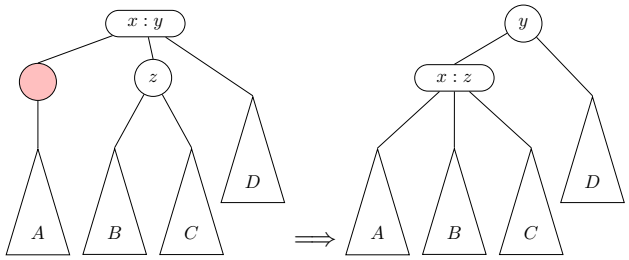
(2) If the parent is a 2-node and the sibling is a 3-node: Split the sibling and rearrange the sibling and parent keys appropriately to create two 2-nodes on the sibling's level and attach all four subtrees appropriately. The hole vanishes and we are done. Important! Consider the variations on the arrangements!

Definition 6.1.2. This is called a *key rotation* because we are rotating a key from a sibling through the parent. This same operation with the same name will appear in B-trees. Note that this is not the same type of rotation that we saw in AVL trees.



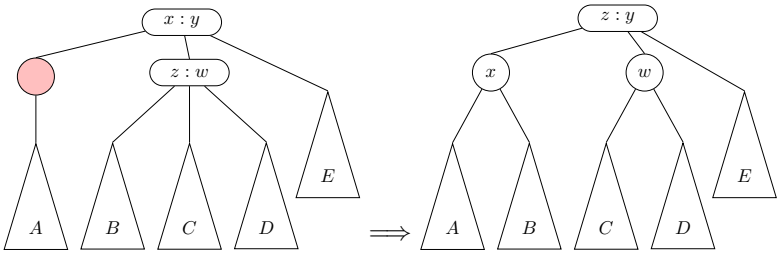
(3) If the parent is a 3-node and one sibling is a 2-node: Demote the appropriate key from the parent and the sibling to make the sibling a 3-node and attach all three subtrees appropriately. The hole vanishes and we are done. Important! Consider the variations on the arrangements!

Definition 6.1.3. This is called a *merge* because we are merging the 1-node with a sibling (and demoting a key from the parent). This same operation with the same name will appear in B-trees.

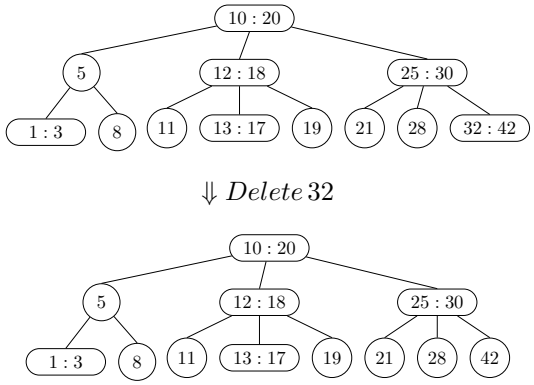


(4) If the parent is a 3-node and one sibling is a 3-node: Promote the appropriate key from the sibling to the parent and demote the appropriate key from the parent to the hole and attach all four subtrees appropriately. The hole vanishes and we are done. Important! Consider the variations on the arrangements!

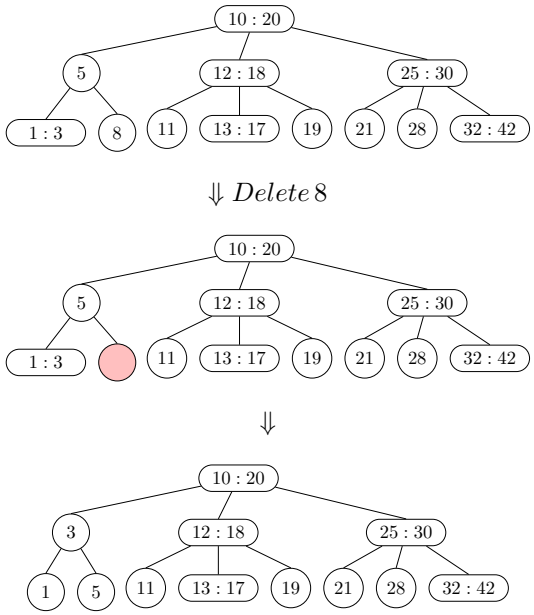
Definition 6.1.4. This is called a *key rotation* because we are rotating a key from a sibling through the parent. This same operation with the same name will appear in B-trees. Note that this is not the same type of rotation that we saw in AVL trees.



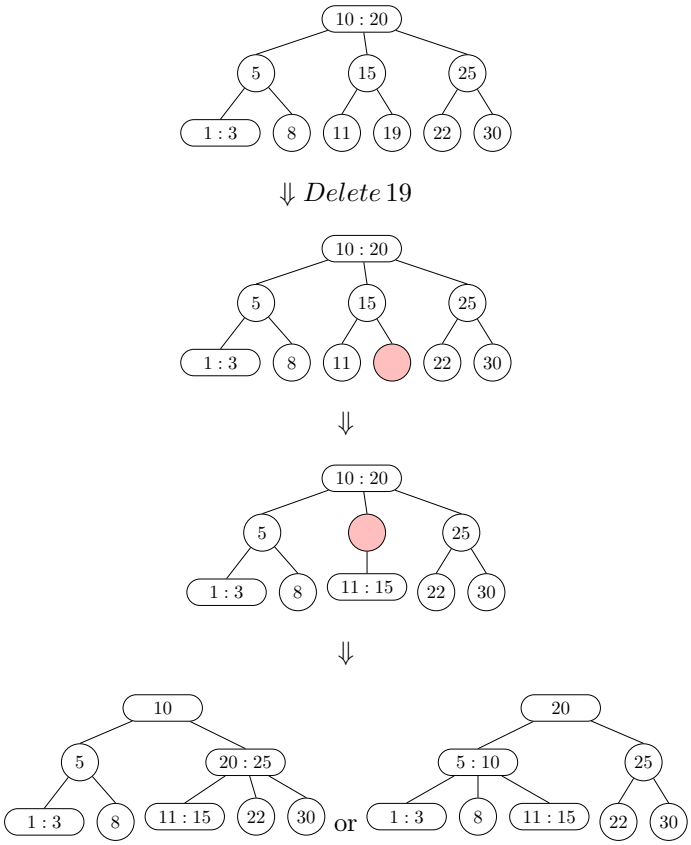
Example 6.1. Here is deletion of the key 32. As we see, the 32 simply vanishes:



Example 6.2. Here is deletion of the key 8. Since the parent is a 2-node and the sibling is a 3-node we act accordingly and we are done:



Example 6.3. Here is deletion of the key 19.



6.2 Time Complexity

The four processes are each $\Theta(1)$ but because of the first case the process might propagate to the top of the tree and so consequently the result is $\Theta(\lg n)$.