CMSC 420: Binary Search Trees

Justin Wyss-Gallifent

February 8, 2024

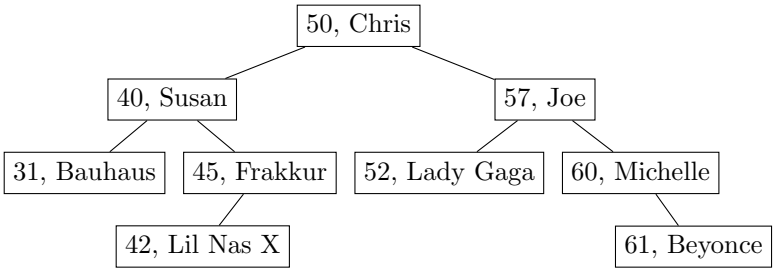
1 Overview

Definition 1.0.1. A binary search tree (BST) is a binary tree, meaning each node has 0, 1, or 2 children. Each node has both a key and a value.

The nodes are arranged such that an inorder traversal yields the keys in increasing order.

More specifically, for any given node with key k the left subtree of a node contains only keys less than k and the right subtree of a node contains only keys greater than k.

Example 1.1. Here is a binary search tree that contains the ID numbers (double digit numbers) and names of some students:



Note 1.0.1. For ease of study typically our keys will be unique positive integers and we will not bother with listing or discussing the values. One could think, for example, that the keys are student IDs and the values are student names, or entire records.

In reality the keys can be members of any totally ordered set, loosely meaning any set such that we can compare and order any two elements within the set. This means we could use integers, real numbers, or words with alphabetical ordering.

2 Search

2.1 Algorithm

Finding a particular key in a binary search tree is easy. We start at the root node. If that key matches, we're done. Otherewise if the target key is smaller, we follow the tree left, and if the target key is larger, we follow the tree right.

We either find the key or we fall out of the tree at a root node and the key doesn't exist.

2.2 Time Complexity

In the best case we're searching for the root. This is $\Theta(1)$.

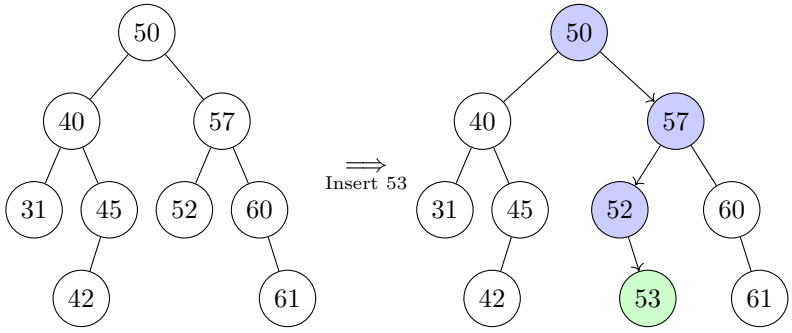
In the worst case the tree is basically a linked list. This is $\Theta(n)$.

3 Insert

3.1 Algorithm

To add a new key to a binary search tree we start at the root and follow the branches by comparing the new key to the keys at the nodes, just as if we were looking for the new key. When we fall out of the tree at the leaf, this tells us where we should put the new node and so we simply attach it at the bottom.

Example 3.1. Given the tree from earlier, with the names suppressed, let's insert the key 53. We start at the root and follow the appropriate path. In the figure below we've highlighted the trip through the tree in the first picture and then the attached node in the second:



3.2 Time Complexity

In the best case the tree is basically a linked list and we're inserting on the other side. This is $\Theta(1)$.

In the best case the tree is basically a linked list and we're inserting on the linked list side. This is $\Theta(n)$.

4 Delete

4.1 Algorithm

Deleting a key is more challenging. The approach is as follows:

1. If the key is at a leaf node then it's easy, we just throw out that node.

2. If the key is at a node with just one child then it's easy, we remove that key and promote the child (and its entire subtree)
3. If neither of these are true, then it's a bit tricky. First we'll find a replacement key from further down the tree. We can use either the smallest key from the right subtree (go right once, then left as far as possible, if at all - this is the inorder successor) or we can use the largest key from the left subtree (go left once, then right as far as possible, if at all - this is the inorder predecessor) We take this replacement key (well, entire node) and we move it up to our deleted node. Now we need to fill the hole left by the replacment node. However by contruction the replacement node was one of the first two types, so it's easy and we're done.

Note 4.1.1. Why do these choices work? Well in general the replacement key needs to be larger than everything else in the deleted node's left subtree and smaller than everthing else in the deleted node's right subtree so that when we replace it, the properties of a binary search tree are preserved.

This means choosing finding the node whose key is either next smallest from the deleted node's key or next largest from the deleted node's key.

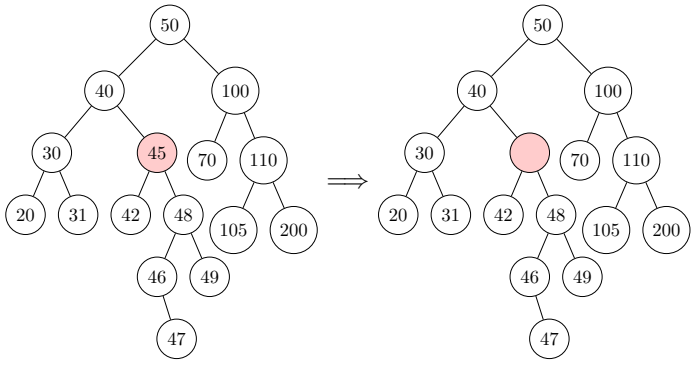
These correspond to our two options.

Note 4.1.2. Convince yourself of two things:

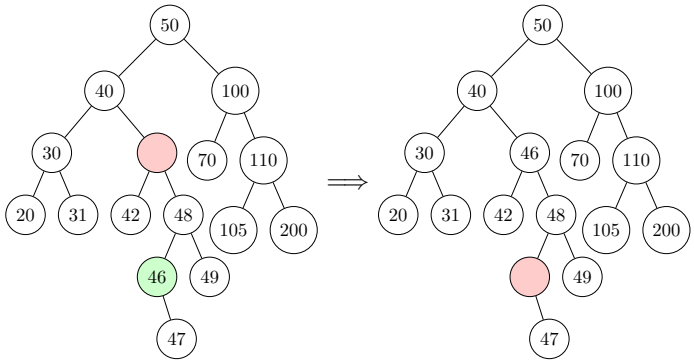
- Using the inorder successor as a replacement preserves the binary search tree nature of the tree.
- The inorder successor is the leftmost key from the right subtree.

Also convince yourself that the inorder predecessor would work, too.

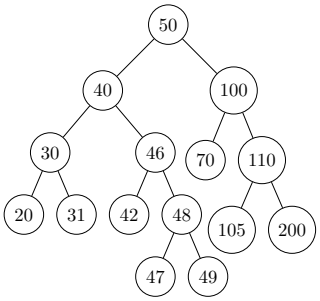
Example 4.1. Let's delete the node with key 45 from the following tree. First we throw it out, leaving a hole:



The inorder successor - go right to 48, then left as far as possible - is 46, so we swap 46 into the hole. This leaves a new hole:



But now the hole has one child so we simply promote that child.



Note that if that child had a subtree then it would just go along for the ride.

4.2 Time Complexity

The time complexity of delete is a bit nuanced. The actual deletions and replacements are $\Theta(1)$ but the finding of nodes is the critical issue.

In the best case we're deleting the root. This is $\Theta(1)$.

In the worst case the tree is basically a linked list and we're deleting the bottom node. This is $\Theta(n)$. Note that we might argue that having replacements might make it worse but even then, in the worst case we would find a node, replace it with a node further down, then do a promotion, and all of this is still $\Theta(n)$.

5 Tree Height Notes

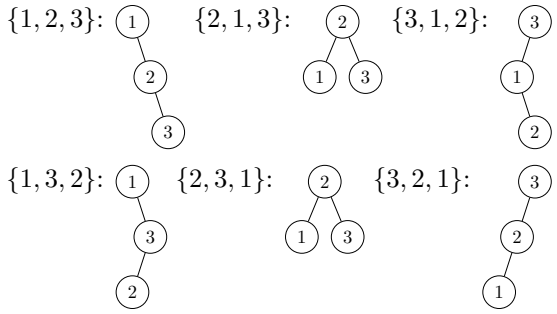
A few notes about the height of a binary search tree with n nodes:

1. It is obvious that in a worst-case scenario the tree can grow to height $n-1$ essentially as a linked list.
2. It is also fairly easy to calculate that in a best-case scenario the tree will have height h where h is the smallest integer satisfying $h \geq \lg(n+1) - 1$, so $h = \lceil \lg(n+1) - 1 \rceil$. This will happen if the tree is a complete binary tree.

3. The average-case scenario is a little more difficult. It can be shown, and this is not trivial, that if n distinct keys are inserted into a binary search tree and if we look at all possible $n!$ possible insertion orders, the expected height is $\mathcal{O}(\lg n)$.

   Some evidence can be easily gathered by picking an n, taking each permutation of $\{1, 2, ..., n\}$ and constructing a binary search tree using each permutation and then averaging all the heights.

   Example 5.1. If $n = 3$ there are six possible permutations yielding the following six binary search trees:



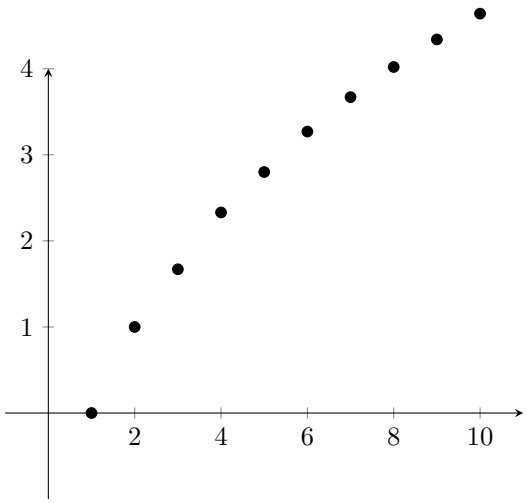
   The average height is:

$$\frac{2+2+2+2+1+1}{6} = \frac{5}{3}$$

   If we do this for various n (work omitted - try it!) we find the following:

n	Average Height
1	0
2	1
3	$10/6 \approx 1.67$
4	$56/24 \approx 2.33$
5	$336/120 = 2.8$
6	$2352/720 \approx 3.27$
7	$18496/5040 \approx 3.67$
8	$161984/40320 \approx 4.02$
9	$1575040/362880 \approx 4.34$
10	$16841600/3628800 \approx 4.64$

   If we plot these we get the following, which clearly appears logarithmic:



4. David Mount's notes contain a proof of a lighter version of this, namely that the leftmost node is at expected depth $\mathcal{O}(\lg n)$.

5. The analysis is more challenging if we allow deletions. A sense of "average case" can be imparted by assuming we make a series of insertions and deletions such that there are an average of n nodes in the tree. It turns out that the expected height is not $\mathcal{O}(\lg n)$ but the worse $\mathcal{O}(\sqrt{n})$. This turns out to be caused by the fact that in deletion we typically use the inorder successor and this causes a less balanced tree. Evidence (but no proof) suggests that randomly choosing between the inorder successor and the inorder predecessor resolves this.