MTH 4300, Lecture 23

Binary Search Trees; ;
The Ups and Downs of Binary Search Trees;
Removing a Node from a Binary Search Tree

E Fink

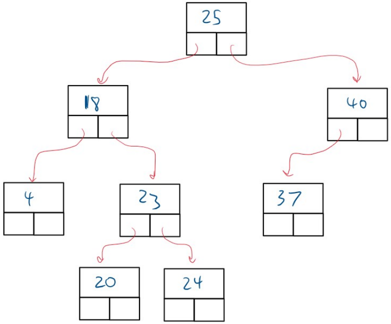
May 5, 2025

1. Binary Search Trees

A binary *search* tree is a binary tree where the following property holds:

> *The data in each node is greater than or equal to the data in all the nodes of the left subtree and less than all the data in the nodes of the right subtree.*

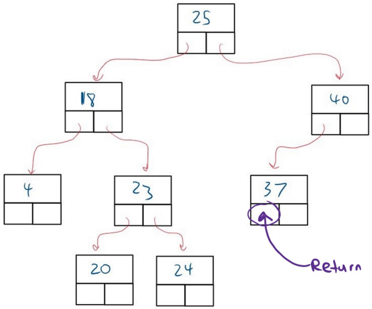
For example, the following illustrates a proper binary search tree:



Binary Search Trees

Searching for the presence of a value within a binary search tree is relatively simple:

- Compare the tree's root node data to the value sought.
- If the value is less than or equal to the root's data, search the left sub-tree, otherwise search the right sub-tree.
- If you either find the data sought or hit `nullptr`, return a pointer to the Node in question (by reference – this will be helpful for later).

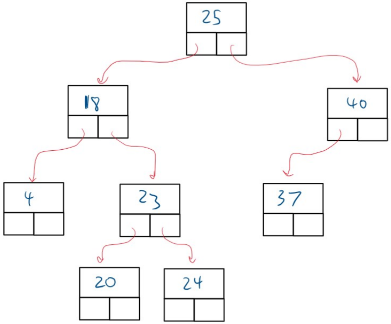


For example, if we searched for 30 in this tree, we'd look at 25, then go right; then we'd look at 40, and move left; then we'd look at 37, and move left; and then we'd hit a `nullptr`, so we'd return the indicated pointer by reference. Of course, the value of that pointer will be `nullptr`.

L23x1_bst.cpp

Binary Search Trees

What about inserting an entry? Well, for starters, where would 21 go in this tree? What about if another entry equal to 37 were added?



Binary Search Trees

The insertion function should:

- Compare the tree's root node data (if the pointer to the root isn't `nullptr`!) to the value which we wish to insert.
- If the value is less than or equal to the root's data, insert within the left sub-tree, otherwise insert within the right sub-tree.
- When we hit a `nullptr` value as the sub-tree you are supposed to search through, create a new node with the given value as data, and replace the `nullptr` value with a pointer to that node.

The insertion function will be implemented as

```
void insert_bst(TreeNode<T>* &, const T &)
```

The `TreeNode<T>*` argument is a pointer to the root of the tree which we insert into. This argument is passed by reference because this function will modify the tree that is passed into it.

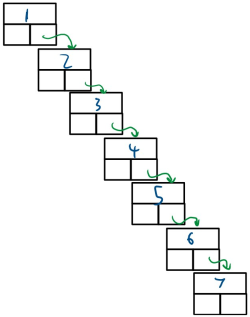
2. The Ups and Downs of Binary Search Trees

Binary search trees can, in theory, allow fast searching and fast insertion of nodes (and removal, as we'll see). That's because the amount of time necessary for both these operations is roughly proportional to the **height** of the tree.

Fact: a binary search tree of height h can hold roughly 2^h entries. That means that if you wish to hold N entries, the height *can* be as little $\log_2 N$. And $\log_2 N$ is a relatively small number.

The Ups and Downs of Binary Search Trees

So what's the downside? The downside is that word *can*. Imagine that we have a tree where we add the entries 1,2,3,4,5,6,7 in that order. What would that tree look like?



If the tree is *unbalanced*, we lose the “low height” advantage.

3. Removing a Node from a Binary Search Tree

L23x2_remove.cpp

We want to write a function that will receive a pointer to a node as input; the function will then remove the node from the binary search tree, in a reasonably efficient manner, so that it leaves the remaining nodes *still arranged in a binary search tree*. This is tricky.

There are three cases: the node to be removed has no left child (perhaps no children at all); the node has no right child; and the node has two non-`nullptr` children. (We also check if the node to remove is `nullptr`.)

The first two cases are simple enough:

- If the node to be removed has no left child, update the pointer to this node to a pointer to the node's right child (which may be `nullptr`), and delete the node itself.
- If the node to be removed has no right child, update the pointer to this node to a pointer to the node's left child, and delete the node itself.

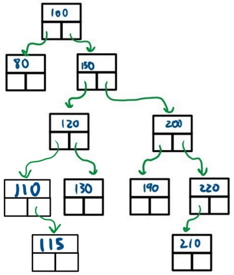
Let's visualize what would happen if we removed the nodes for 20 or 40 from the tree we had before (see a couple of slides earlier).

Removing a Node from a Binary Search Tree

The last case, where the node we wish to remove has two non-`nullptr` children, is the trickiest.

The *successor* to a node is the node in the tree which has the next greater entry. This is actually easy to find: it will be the *left-most node in the right sub-tree*.

For example, where are the successors to 100, 200 and 220 in the following image? (One of these might be a trick question.)



Removing a Node from a Binary Search Tree

The punchline of the “2 children” is that when the node to be removed has 2 children, it should be **replaced by its successor node**.

However, this is easier said than done, and in fact there are two subcases: when the successor is the immediate right child of the node we wish to delete, and all other cases. It’s not immediately obvious why we need to separate these!

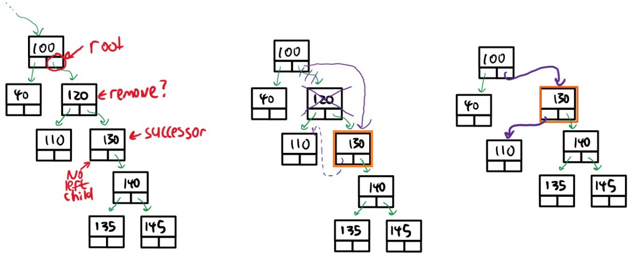
Removing a Node from a Binary Search Tree

First, let's deal with the case where the Node we wish to remove has its immediate right child as a successor.

We'll say that `root` is the pointer to this Node. This case isn't so bad:

- Let `succ` be a pointer to the successor. In this case, `succ->left` should, originally, be `nullptr`. Simply reassign this to be `root->left`.
- Now, we delete `root` ...
- ...and reassign `root` to be `succ`

For example, in the below, here's what removing the node whose data is 120 would look like. (`root` would be 100's right pointer, which points to the node with 120 initially; and `succ` would point to 130.)

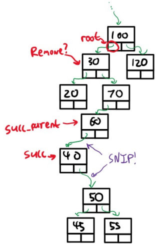


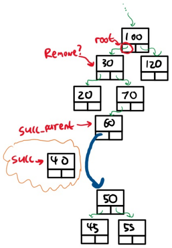
Removing a Node from a Binary Search Tree

Finally, the subcase where the Node we wish to remove does NOT have its immediate right child as a successor. Again, let `root` be the pointer to this Node.

- Locate pointers to the successor (call this `succ`) AND the successor's parent (`succ_parent`). Note that the successor will be its parent's left child.
- The successor itself will have a `nullptr` left child (why?), but its right child may or may not have a null right child. *Take the right child of* `*succ`, *and make it the left child of* `*succ_parent`.
- . . .

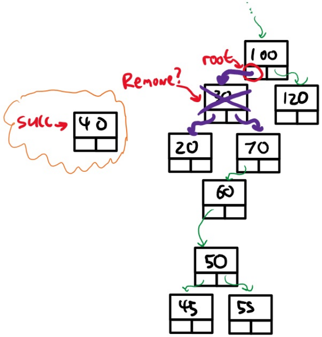
In the below, we illustrate the first part of removing the 30 node from the tree.

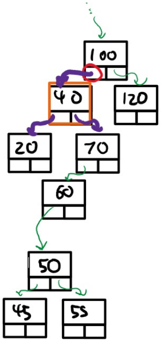




Removing a Node from a Binary Search Tree

- . . . Now, set the successor's left and right pointers to be the left and right children of `*root`.
- Finally, we can delete the node currently pointed to by `root`, and replace it with the successor.





In the code, notice how we use use the `search_bst()` function to locate the pointer within the tree containing a value we wish to delete. This works because `search_bst()` returns by reference – it will return a reference to a part of a node within the tree.