Data Structures with C++ : CS189

Lecture 7-2: Binary Search Trees

Recap

- A tree gives us the ability to look up data in log time instead of n
- To find data, it starts with a pointer to the root, and travels left to find smaller data, and right to find larger data
- The basic BST doesn't allow duplicates
- STL does not offer a BST container, but STL's "Set" is mostly the same

Recursive?

- A tree does not automatically mean recursion
- The guiding principle is if an algo needs to fork, it is recursive. Otherwise use a loop.
- There is exactly one path from the top to what you are looking for, so it doesn't fork
- Something that *needs* to touch every node (copy) is recursive because it forks

Different Implementation

```
struct VectorTreeNode {
	T mData;
	int mLeftIndex;
	int mRightIndex;
};

class BST {
	vector
	 <VectorTreeNode>
	 mNodes;
};
```

- You don't know how STL implements Sets
  - Information Hiding!
- It happens to use a red-black tree last time I looked it up
  - Variations of trees are later.
  - Red-black is "better but slower" and "hella hard"
- Why not a Vector?
  - Vectors can't shrink well
  - That's how Heaps work later on though
- Pointers and indexes both mean "Where can I find the next one?"

Deleting Nodes

Note:
In the next few slides there are mentions of Double Pointers. They were causing too much trouble and they weren't the point, so I replaced them with the Parent pointer.

- Draw a picture of a tree
- Deleting a leaf is easy, but what if there are two children?
- Algorithm is this:
    - No children? Delete it
    - One child? Whomever was pointing at me now points at my child. (Skip me.) Now delete.
    - Two children? Don't delete the node! Find the next highest piece of data, swap data, and delete *their* old node. (Which might have a right child.)
        - "Next highest" = Take one step right and then as many lefts as you can. "The smallest data bigger than us."

Now a Set

```
struct SetNode {
	T mData;
	TreeNode *mLeft;
	TreeNode *mRight;
	TreeNode *mParent;
};
```

- One more time, tree is the tech, Set is the ADT
- The only difference is that to make remove easier, the node gets an "up" pointer
  - Breaks scientific definition of tree
- Without being able to go up, we need to have two walkers so we can stitch the tree back together
  - Or *shudder* Double-pointers
- Being able to go up also lets us go left and right
  - This is how Set's iterator can have a Next.

Map

```
struct MapNode {
	K mKey;
	V mValue;
	TreeNode *mLeft;
	TreeNode *mRight;
	TreeNode *mParent;
};

map<string,int> tAges;
tAges["Bob"] = 42;
```

- A Map is a Set, but with two pieces of data in each node
- The Key is used as the search term
- The Value is used for the resulting data
  - We saw this usage in the "Use STL" week before
- Remember, maps have two separate template types
  - Only the Key must have < or a Functor.
- Almost every place we had a T, we now have a K
  - The only place V shows up is as return values

Map/Operator Rant

```
map<string, int>ages;
ages["Bob"] = 4;
if( ages.find("Bob") !=
        ages.end() )
        cout << "Yes";

if( ages.find("Alice") !=
        ages.end() )
        cout << "No";
int Y = ages["Alice"];
if( ages.find("Alice") !=
        ages.end() )
        cout << "YES!?!";
```

- I personally think Maps are the coolest
- But in order to look like an array, it uses operator [], which returns a reference like other containers' accessors
- But if you access a map with a key that is not there, the key GETS ADDED
    - The gettor is NOT CONST?!?!
- No other language does that
    - There is a method called At that does a get but without that horrible horribleness
        - It's not really a bug, it's just how [] works
        - One reason I hate operator overloading
            - A * B != B * A is another

End

Anybody who leaves early and doesn't get an A is a silly person