CS280 Notes 2/22

by Christian Sagel

Midterm will cover everything up until last week. Today's topic won't be covered.

Trees (Part 2)

The main operation performed on trees will be rotating them.
We will start with an AVL tree, easy to understand implement.
The worst case for searching is now O(log N) which is very good.

when we insert, we always insert at the bottom of the tree. If we determine at that point that the tree is still balanced, we don't have to do anything.
If we determine if the node makes it go out of balance, we have to find the offended node. We have to go back up the tree. When we find a node out of balanced, we have to rotate to fix it left or right.
When we delete, we have to go all the way down.

BSTs are not balanced. Sorted data messes it up.
On an AVL we will have a stack.
The balancing algorithm is not recursive, we just have to go up the stack.
to be balanced the height between the subtrees must not differ by more than 1.

Rotation same concept as promotion. When you say rotation, you say what direction.
Rotations preserve the order.
We will be allowed to use the STL stack to do the AVL implementation.
You should not have the insertion or deletion balance the tree. They are separate issues.

As you walk down the tree you will be pushing pointers to the stack.
A balance method to do the rotations.
You will lose different points if you use different algorithms for insertion and deletion.
Instead, use a flag so you don't repeat your code. Put a check in that function, don't do two separate functions.
Checking the height all the way up the tere will be expensive.

All you need to know is which is heavier on the left or right. Instead you can store a balance factor on the tree. Nodes heavy on the left are -1, 0 if balanced, heavy on the right -1. If you have a +2 or -2 that tells you.

You don't have to keep balance factors. With rotations, they get too expensive to keep.
You don't have to do balance factors for the assignment (they were extra credit previously) because they can get very complicated.
You have to implement balance factors for a real-world implementation.
For a lot of data that you have to search quickly you will need a tree.
Deleting from a red-black tree it takes 3 times as long as to do the delete.

Assignment
Don't have to use balancing factor.
Keeping the counts in the node is extra credit allows us to implement a subscript operator without having to walk and count the nodes.
We will be using an object allocator so we don't allocate one node at a time.
There's a flag to decide whether to share the object allocator among all instances of the tree. Since they are all of the same type.
We do not want to make copies of the object allocator. It was disabled for the copy constructor.
If doing extra credit return true for ImplementedIndexing();

We will be implementing a BST class first, then inheriting from that to make the AVLTree.

The insert/removes for the AVL tree will be different than the BST. The other methods we will inherit.
He recommends using his DummyOA for the assignment.
Starting with the templatized version is a lot of work. Mead is giving us a sample driver that allows us to work with integers first rather than going straight to the template version. Starting that way will be much more more manageable.

There's a spellchecker driver for stress testing. Every word, a char pointer, is a node on the tree. 20 comparisons for a million items. A great speedup!!

new: allocates memory and calls the constructor
delete: calls the destructor and frees the memory

placement new: Passing an address after new calls an overloaded version that just . To construct objects on memory that you already own.
To free a node: Call the destructor explicitly, then pass it to the object allocator.
95% of the BST class is already posted on the assignment. The point of the assignment is the balancing algorithm on the AVL tree (even if we are given the pseudo code).
There's a 1:1 correspondence with the pseudocode.

The balance factor is hard.
The assignment is due in 2 weeks.

Try doing the AVL self-check!

2-3 Search Trees
Another kind of tree.
BTrees are trees that are huge, bigger than memory. For very large data sets. They are stored on the harddrive, used by filesystems. Databases.
Each node can contain 1-2 keys.
We are inserting nodes to keep balance. We don't check nodes, or rotate subtrees to keep the balance.

2-3-4 trees the smallest kind of BTree.
The more children you have the more breadth rather than depth you have.
Every ndoe has at most 4 children. Each noca can contain 1.2 or 3 keys.
Splitting proactively will make it so we only have to go up only once to the parent node.
Rotations are done to preserve the sort order. Instead we will be passing keys around.
Splitting the root is what causes the tree to grow one level.

The red-black manages what to do with the node, a flag to determine the deletion function. For most algirhtms the delete function is the most complicated.
THere are 7 different cases in the 2-3-4.

Thrown sorted data, you will get all 2 nodes.
Best case is log(4).
With the delete algorithm we merge on the way down. Merging on the way down.