Programming Project 9

EE312 Spring 2014

Due April 21, 2014 before the end of the day

FIVE POINTS

General: ONE MORE TIME (with feeling). Once again, we're going back to the CRM example, this time to apply our newly minted knowledge of the Binary Search Tree. You'll build the CustomerDB struct to perform almost the same functions as last time, but instead of using an array to hold the customer data, you are required to construct a binary search tree.

Your Mission: Your work for this project will be inside CustomerDB.cpp. I'm providing you with a solution to Project8 (called Project9-solution.cpp). You should not have to make any changes to this file! You are also not permitted to make changes to any of the .h files used in this project.

Step 1: The customer tree

I've gone ahead and created a TreeNode struct for you to use to build your binary search tree. Note that in Project8/9 we never need to remove nodes from our tree. Well, that's not quite true, we need to remove nodes when we clear the database or when we run the destructor at the end of the main program. But we never need to delete just a single value from our tree – if we're deleting, then we're deleting the whole tree. That's cool, 'cause that means we don't have to worry about the remove function. We'll be continuing the use of the [] operator that we used in Project8, and the operator will function in basically the same way – if you can find the right customer in the database, then return a reference to that customer. If you can't find the customer, then add the customer to the database, initialize the customer correctly (zero bottles, rattles, and diapers), and return a reference to that customer. Of course, when you add a customer to the binary search tree, we need to add it into the correct spot in the tree. But, it's a fairly simple matter to copy/past/edit the BST code from lecture to make this work with your tree.

There's one important, and subtle issue with our binary search trees in this project that I need to explain. Since our tree is organized (sorted) by the customer names, it would be extremely uncool if someone changed the name of a customer after that customer were placed into the tree. In a tree with Craig at the root, Billy to the left and Sue to the right, we're all good. But if someone changes the Customer struct stored in the root so that the name is Allen, then we'll never be able to find Billy in the tree (changing the name will violate the binary search tree property). Since changing a customer's name is so problematic, I've taken advantage of some C++ functionality to make it impossible to change a customer's name. I've declared the name component in the customer struct to be a "const String". That means, once the String is initialized with a value, that value can never change. There's comments talking about this in the code, and you probably shouldn't be trying to change the names of customers anyway, but it is important that you

recognize that changing the name of a customer will ruin the tree unless we remove and reinserting the node into the tree.

Anyway, write your operator[] so that it performs the same hybrid "find" and "insert" operations that it has done for us in the past. If you end up doing the insert functionality, you'll be creating a new TreeNode to hold the customer. I've taken the liberty of declaring and initializing a variable (called "leaf") that you can use if you end up needing to insert a new node.

Step2: FindMax and scanning the tree

Binary search trees are great data structures, but they can't do magic. In this project, we're going to be searching a tree for lots of different things, and a tree can be organized around only one component. So, we organize our tree around the Customer's name. That makes it easy to find the customer we're looking for, but not so good when it comes to finding the customer who's purchased the most bottles. So, we're stuck having to scan the entire tree. In this project, I'm asking you to build functionality that will make it possible to scan the database. I'm using something of a kludge between the C++ conventions for scanners and the Java conventions for scanners. Hopefully the behavior I'm asking for end up being more intuitive to you than either a C++ iterator or a Java Iterator.

Here's how it works. The client code (e.g., Project9-solution.cpp) declares a variable of type TreeScanner and initializes that variable by calling the CustomerDB::startScan() function. The client can then use this scanner to look at each customer in the tree. To look at the current customer, the client calls the get() function on his scanner object. To advance to the next customer, the client calls the advance() function on his scanner object. Finally, to determine if the client has finished scanning the whole database, the client calls the done() function on his scanner object. The client-side code for doing all these things in Project9 has already been written. If you look at the findMax function inside Project9-solution.cpp you'll see how the TreeScanner is used. Your job is to implement the scanner. The get() and done() functions have already been written. The advance() function and the startScan function you'll have to write.

Step 3 – Cleanup and Testing

Make sure your destructor and clear functions work correctly (I already wrote the destructor, make sure you write clear correctly – you might want to test this function!!). Hopefully this is pretty easy.

Testing should be relatively straightforward. There aren't supposed to be any bugs in Project9-solution.cpp. So, if you implement the CustomerDB functions correctly, the program should pass all the Project8 tests just fine. You may want to test your operator[], and the clear() functions directly (write your own main, and don't bother with reading an input file).

Step 4 – Optimizing the Tree

The “bonus point” for this project will be dedicated to completing Step 4. In Step 4, you must write two functions, CustomerDB::height(void) – return the height of the tree, and CustomerDB::rebalance(void) – rearrange the nodes in the tree so that the tree has the minimal possible height. Both of these functions are huge opportunities for you to practice your recursion skills. Do keep in mind that it is easiest to code recursively if you write a separate function (a normal C function, not a member function) that does the recursion. Just have your member function (height and rebalance) call the recursive function with the correct parameter. There’s a few hints inside the source code. Good luck!