CSE331 – Project 5: Balanced BST

In this project you will complete an implementation of the AVL balanced binary search tree data structure and use it to store and manipulate data from the Internet Movie Database. The project must be submitted to Handin no later than 11:59 pm Friday March 1, 2013. Email the TA with any questions. Good Luck!

Project Description

You have been given code to implement a program that will load and manipulate the data from the Internet Movie Database. The program accepts three arguments from the command line: (1) the imdb datafile, (2) the number of actors (or -1 to read the entire list) and (3) a file containing the list of commands. The executable accepts the following commands:

1. height – prints out the height of the BST
2. actor-movies – list all of the movies for a given actor, the actor is given as an argument
3. movie-actors – list all the actors for a given movie, the movie is given as an argument
4. delete-actor – delete an actor from the database, the actor is given as an argument
5. delete-movies – delete a movie from the database, the movie is given as an argument

All commands occupy a single line in the file, as do their arguments. An example command file is provided, called test.txt. The code to load the database and handle these commands is already written for you. When the actors and movies are completely loaded, they will contain cross-references to one another. When an actor is deleted, it's references are also deleted. When all the actors from a particular movie have been deleted, that movie is also deleted. The reverse is also true for movies and actors.

Your job is to complete the Tree.h file which has been provided. You must provide an efficient implementation of an ***AVL balanced BST***. Most of your functions should have a best case running time of O(log N), except for preorder, which will always take O(N), since it must recurse over every node. This project is very similar to Project 4, however you may use lazy deletion. Also we will be keeping track of the height of each node with a private variable in the node class.

You may not use anything from the STL for this project. You may use IO for debugging, but be sure to remove debugging IO from your completed project. You will be marked down for any debugging IO left in your project. As always you may not change the signatures of the public methods, the node class, or the way in which the tree class interacts with the rest of the program. You may add any private member variables or methods to CTree to help you complete the project. You may also assume that any templated types passed to CTree will have overloaded output stream operators (aka “<<”).

Programming Notes

For this project you will *need to implement your AVL tree commands using pointers to pointers.* This is done to preserve several assumptions used by the imdb driver program. You should still be able to use the code in the book as a guide, just remember that we are using pointers to pointers instead of references to pointers. Also, here are a few tips for working with a pointer to a pointer:

```
//gets a pointer to the root pointer
Node **p_tree = &m_root;
//dereferencing p_tree once gives us a pointer to a node
Node * ptr = *p_tree;
//twice gives us access to node itself
cout << (*p_tree)->Data() << endl;
```

When you implement *rotations, make sure you do it by changing the connections betweens the nodes. Do not move the nodes around in memory, and do not simply copy the payloads.*

We recommend that you implement all of your functions in the private space, and then place calls to private methods in the public methods. This will reduce the amount of code you write when overloading the const methods. We highly recommend that you use the private space for implementing rotations! Remember that double rotations may be accomplished with two calls to a single rotations, this lends the implementation of rotations to functional abstraction.

A sample database file is provided in the project directory, this is a subset of the full imdb dataset and it has all of the data necessary to work with the sample command file. These sample files are provided to help you test primary functionality, you should perform much more rigorous testing on your completed projects and make sure to test them on Adriatic!

When we grade your project 5. We will be giving ~40% based on the primary functionality of your AVL BST on a primitive type. This will be done by generating a random list of >100 data items, pushing them onto the list and then performing a random sequence of finds and deletes. You will receive ~10% if your project works correctly with the IMDB driver.

In the Makefile you will notice a variable at the top of the file, CPPFLAGS. This variable contains the flags we want to pass to the compiler. If you are debugging your code, then you would probably want to change the flags to: -g -Wall. We will be compiling your projects with the flags: -g -O3.

In the directory ~cse331/Projects/Examples, we have placed a windows program that you can use to explore the behavior of trees, both balanced and unbalanced. The executable is called “VisualTree.exe”.

Project Deliverables

The following files must be submitted via Handin no later than 11:59 pm Friday March 1, 2013:

- Tree.h – contains your implementation of a templated doubly linked list
- project5.pdf – files containing your answers to the written questions. (Must be a Portable Document Format file)

Written Questions

Submit the answers to these questions in a PDF file, called project5.pdf, along with your source code files. Any images or diagrams must be embedded in your PDF file, they will not be graded if they are separate files. For problems asking for a short paragraph, the paragraph should consist of 4-5 complete and concise sentences.

1. What are the heights of your movie (h_m) and actor (h_a) trees when you load the entire data set? Given the heights h_m and h_a, how many nodes can be stored in each tree, how many nodes are actually stored in these trees? In a short paragraph, describe how this differs from the results in your project 4.
2. Assume that you are given a binary search tree which uses lazy deletion.
    a. Give a pesudo code algorithm for a new private member function in our tree class. This function must rebuild the BST in O(N) time. Assume that you are given the following private member function: `vector<Node*>* inOrder()`. Where inOrder returns a pointer to a vector containing pointers to all of the nodes in the tree, in sorted order (obviously, this is the result of an in-order traversal). Assume that inOrder also deallocates the “lazily” deleted nodes and runs in O(N) time.
    b. Give a good heuristic for deciding when to rebuild our tree without the “lazily” deleted nodes. Explain why your heuristic is reasonable, and how it effects the asymptotic bounds on both space and time. NOTE: a heuristic is simply a rule, it need not be perfectly optimal, but you should be able to explain in what ways it will impact time and space management.
3. Write a pesudo code algorithm to perform an in-order traversal of a BST that does not use a recursive function call (i.e. the algorithm must be contained within a single looping structure in a single function). HINT: What data structure do you know of that will preserve the state variables in a recursive fashion?
4. Give a big-Theta bound on the running time for your algorithm in number 3. Justify you answer with a short paragraph.