EECS 2510 – Nonlinear Data Structures & C++ Programming Spring 2016

Programming Lab #3 – Due Thursday, March 31, 2016 @ 8:30 AM

This lab wraps up our programming study of binary search trees. For this lab, you will implement all *three* binary search trees (BST / AVL / RBT) , compare their performance, and submit a report

1) Code the "standard" Binary Search Tree. You have already written this code in a previous lab.

2) Code the AVL tree. You do not have to implement all seven dynamic set operations. All you *must* have is the Insert and (perhaps) Search routines (and an in-order traversal, but that should be trivial at this point). Recall my caveat about the "symmetric" aspects of the "else" code for AVL_Insert – don't just blindly replace "left" with "right" and vice-versa, and remember that balance factors of -1 are symmetric to balance factors of +1. Carefully study the schematic diagrams for the rotations, create their symmetric analogs, and determine for yourself where you need to swap "left" and "right" (and where you don't).

3) Code the Red-Black Tree, as far as Insert and Search (yes, and a traversal). Note that "Insert" will require the "insert-fixup" and rotations to be implemented, but you have the pseudocode. You don't need Delete (or its fixup).

For the RBT, you will need to define something in your class like `node *nil` in addition to the usual `node *root`. In your constructor for the tree, instantiate the `nil` node to be black, containing the empty string (""), and have its parent and children pointers all point to itself `(nil)`. Create the root of the tree to point at the `nil` node. If you start from this point, I believe you won't have any trouble implementing the RBT code. This approach does NOT solve all of the implementation problems for deletion, however, so do not use this code as the basis for a full-featured RBT implementation – it would cause problems if you were to attempt to implement the delete functionality and try to re-balance the tree.

For all three trees, you must also code a private method to return the height of the tree. Your code should also expose a public method to print the height of the tree (which will call the private method to actually compute the tree's height). Think about how you built the Huffman strings. That might help you with determining the actual height of a tree.

You will need to support "inserting" an item that's ALREADY in the tree. In order to accomplish this, simply add an integer field called `count` to each node, like you did for the BST program. When you attempt to insert a string into the tree, if that string is already found, then increment the `count`; if the item is not found, then insert the item as usual, setting the new `count` to 1. This may mean a change to the RBT Insert code, which assumes you've already created a new node to insert – if the word you have is already in the tree, then you don't need a new node.

You will also need to maintain a class-wide variable (for each of the three trees) for the number of search-based key comparisons made. Every time you compare a node's key against what you're inserting (whether in the insert or search routines), increment this count. Print out this count for each tree. This will provide a partial measure of the comparative efficiency of each tree.

Similarly, whenever you change a node's child or parent pointer (AVL won't need parent pointers; RBT will), keep a count of that. For AVL, keep a count of the number of nodes in which you change the BF (after inserting it). For RBT, keep a count of the number of re-colorings.

All combined, the number of node key comparisons, child pointers changed, and other node updates (recolorings / BF changes) done during insert should be a measure of the total amount of work done for each tree.

You will need to modify your `InOrder` traversal code for each tree such that it computes the total number of items in the tree, both unique, and with duplicates – i.e., you will produce a count of the number of *distinct* items in the tree, and also the number of items counting duplicates (this will involve the `count` field). For example, there may be 100,000 *total* words, drawn from a total vocabulary (nodes in the tree) of 10,000. THOSE counts should match across all three trees.

As discussed in class, you may want to test your code with something like the following. This will provide you with a source of some worst-case statistics for the trees:

```
char c;
RBT RBT_T;  // instantiate each of the trees
AVL AVL_T;
BST BST_T;

char chari[10];
for (int i = 0; i<10; i++) chari[i] = '\0';

for (int i = 1001; i<=9200; i++) // insert 8200 strings in BST
{
    sprintf(chari, "%4i", i);
    RBT_T.RBT_Insert(chari);
    cout << "Tree Height is now " << RBT_T.TreeHeight() << endl;
}

for (int i = 1001; i<=9200; i++) // insert 8200 strings in AVL Tree
{
    sprintf(chari, "%4i", i);
    AVL_T.AVL_Insert(chari);
    cout << "Tree Height is now " << AVL_T.TreeHeight() << endl;
}

for (int i = 1001; i<=9200; i++) // insert 8200 strings in RBT
{
    sprintf(chari, "%4i", i);
    BST_T.BST_Insert(chari);
    cout << "Tree Height is now " << BST_T.TreeHeight() << endl;
}

cout << "\n\nPress ENTER to exit\n";
cin.get(c);
```

Inserting values starting at 1001 ensures that all of the values are four-character strings. By inserting approximately 8200 values (in order), and based on what we already know about the height of a degenerate BST, an AVL Tree, and an RBT, you should be able to compute what the approximate height of the three trees should be before you ever run the code. If you would like to tax your trees more strongly than the 1001-9200 loops above, try 10001-26400 (or 10001-75536)

Knowing what the tree heights *should* be may help you determine if your rebalancing routines are working properly. If you expect a tree height of 20 and have a height of 300, then you're probably not detecting imbalances properly and/or not executing your rotations properly.

Finally, you will read the `Shakespeare.txt` file four more times, once as a "dry run", and three for placing the individual WORDS into each of the three trees. This can be done with the following code (there are other, more efficient ways, but this is sufficient):

```
char c;
RBT RBT_T;
AVL AVL_T;
BST BST_T;

char chari[50]; // assumes no word is longer than 49 characters
int iPtr;
bool IsDelimiter = false, WasDelimiter = false;
for (int i = 0; i<50; i++) chari[i] = '\0';

ifstream InFile;
InFile.open("W:\\Shakespeare.txt", ios::binary);
if (InFile.fail())
{
    cout << "Unable to open input file\n\n"
         << "Program Exiting\n\nPress ENTER to exit\n";
    cin.get(c);
    exit(1);
}

iPtr = 0;
InFile.get(c);
while (!InFile.eof())
{
    IsDelimiter = (c ==  32 || c ==  10 || c ==  13 || c ==  9  ||
                   c == '.' || c == ',' || c == '!' || c == ';' ||
                   c == ':' || c == '(' || c == ')' );
    if (IsDelimiter && !WasDelimiter)   // if THIS character is a
                                        // delimiter and the last
                                        // character WASN'T
    {
        WasDelimiter = true;
        RBT_T.RBT_Insert(chari);          // insert this word in the RBT
        AVL_T.AVL_Insert(chari);          // insert it in the AVL Tree
        BST_T.BST_Insert(chari);          // insert it in the BST

        for (int i = 0; i<50; i++) chari[i] = '\0'; // zero the word
        iPtr = 0;
    }
    else if (!IsDelimiter)  // if this isn't a delimiter, keep going
    {
        chari[iPtr] = c;
        iPtr++;
    }
    else if (IsDelimiter && WasDelimiter)
    {
        // Do nothing -- two consecutive delimiters.
    }
    WasDelimiter = IsDelimiter;   // for the NEXT iteration
    InFile.get(c);
}
InFile.close();
```

In the code above, the three bold lines indicate where you need to make a change. Run the code once (and time it) with NO inserts, once more inserting into BST, again, inserting into an AVL, and a fourth time, using an RBT. By counting the elapsed time of each, and subtracting the "overhead" time from simply making a pass through the file with NO inserting, you can get another measure of the work performed.

After you have put all of the words into the three different trees, perform an in-order traversal of each, computing (and printing out) the total number of distinct words, and the total number of words counting duplicates. Also print out the heights of the three resulting trees, and the total number of key comparisons, node pointer changes, and other updates made while doing the insertions.

Write up a one-page report (more if you like, but 1 page minimum) that details your findings, comparing the results of the performance of all three trees on the Shakespeare file. List what you expected, what you found, any surprises, any particular observations, etc. If you test your trees with lists of varying sizes, some graphs that show some performance metric as a function of N would be helpful. Doing more than the bare minimum here is ***strongly*** encouraged. Make sure you have at least a page of text; if you opt for graphs, don't have a half-page of text and a half-page graph.

Note that you must re-work main so that you read the file four times: First, scan the file, and parse it out one-word-at-a-time, and do nothing with the words. That will tell you how much of main's time goes into the file-reading and parsing overhead. Then make another pass through the file, and put all of the words into the first tree, and then re-scan the file, putting all of the words into the second tree, and then making a third pass to fill the third tree. That way, you can time how long it takes for ***each*** tree (clock time is *also* an indicator of total work done – key comparisons, pointer changes, node updates are all roughly indicative, but aren't necessarily absolutely definitive). Check out C++'s `clock()` function (for which, I believe, you'll want to `#include <time.h>`).

Coding Standards:

1) Each source code file ***must*** contain a brief header comment listing, at a minimum:
   - The file's name
   - YOUR name
   - What this code was written FOR (EECS 2510, Spring 2016)
   - WHEN the code was written (the date)
   - A brief description of what the code is/does, etc. If it's just a header file, then reference what it's a header FOR. If it actually DOES something, briefly mention what/how.

2) Every non-trivial method must have a header block comment following the opening brace, telling what the method is, what it does, what its input/output parameters are, and what (if anything) the method returns.

3) Variable names should be descriptive, with the exceptions of simple loop variables, for which 'i' and 'j' are acceptable.

4) Use in-line comments to describe individual variables and lines of non-obvious code

5) Blocks of code that accomplish a particular task should always have a block comment above them, briefly describing what the code does and/or how it does it. Remember, your comments will be a narrative to the next programmer to pick up this code and have to maintain it. It may even be you, so make sure you will know what your own code does a year from now. Just because it's blindingly obvious when you're working on the assignment doesn't mean it would be so obvious to someone else (or even to you) after enough elapsed time.

6) Braces and indentation should follow the Allman style:

```
if (condition)
{
    then clause
}
```

As opposed to the K&R style:

```
if (condition) {
    then clause
}
```

In cases where the code in the then/else clause(s) is trivial, you may be flexible with the indentation and omit the braces. The following is quite clear and perfectly acceptable:

```
if (key > p->data) p = p->RCH;
              else p = p->LCH;
```

Of course, this code could ALSO be implemented with the ternary operator:

```
p = (key > p->data) ? p->RCH : p->LCH;
```

But this much less clear; coding it this way may make for very efficient runtime code, but may also confuse the next person to read your code. If you take coding style liberties to do something concise (and hopefully, elegant), make extra-sure you're leaving comments that help the next programmer (or your instructor) understand what you've fdone, and why you did it the way you did.

Similarly, for loops that do something trivial, there is no need to indent the body of the loop or use braces:

```
for (int i=0; i<=N; i++) Array[i] = 0; // initialize the array
```

7) You are NOT allowed to use any classes from the STL (Standard Template Library). The idea here is that you create your OWN data structures and supporting components.

8) These standards may be amended from time to time.

Submit to Blackboard a 7-zip (preferred, because it compresses better) or zip (acceptable, if you must) archive of your entire Visual Studio workspace.

If you have questions or run into problems, contact me. Don't use code from ANY source other than the lecture slides or the Cormen text, even as a reference. As I mentioned in class, the AVL code Liang shows in the Java text takes a completely different approach, and trust me, it will confuse you. The AVL code you submit should be a completion of the code I provided in the slides. If you want to read Liang's explanations for how AVL works conceptually, and examine his schematics for the tree, feel free; just ignore his code and how he goes about the rebalancing. For RBT, everything you need should be in the lecture slides and/or the Cormen text (Chapter 13).