CSCI 2275 – Programming and Data Structures
Instructor: Hoenigman
Assignment 6
Due: Friday, October 23 by 5pm

Getting to know the Binary Search Tree

This week we started talking about a new data structure called a Binary Search Tree (BST). The BST is an ordered tree where nodes to the left of a node have values less than the node and nodes to the right have values greater than or equal to the node. This ordering means that BSTs can have very efficient insert, search, and delete operations, as efficient as log(n) if the tree is balanced.

In this assignment, you're going to compare the insert and search operations in a BST with those of a linked list. In Assignment 3, you built an array of words using the Hunger Games text, where each element in the array was a word and the number of times it was found in the book. In this assignment, you'll repeat that process, but instead of using an array, you'll add the words to a linked list and a BST and compare the operations needed to search and insert new words.

Data structures you need to build

Linked List class

Convert your linked list code from assignment 4 to simplify the node and linked list functionality. Your node needs to have the following properties:

```
Struct wordNode:
      string word;
      int count;
      wordNode *next;
      wordNode *previous;
```

You can simplify the code to have just the basic linked list functionality – inserting a node and searching for a node. Your insert function needs to add nodes to the linked list alphabetically, including the head node. You will also need a public method to count the number of nodes in the list, such as *countNodes()*, and a public method to count the total words in the list, such as *countTotalWords()*. You can add a print function for debug purposes, but you won't need to print the list for this assignment.

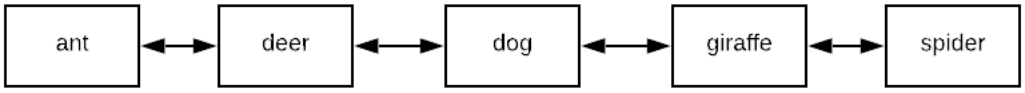
BST class

There is a BST.h header file provided with this assignment with the definition for a BST class that includes basic functionality to insert, search, print, and count the nodes in the tree. You need to implement these functions in a BST.cpp file.

Comparing the BST and the Linked List

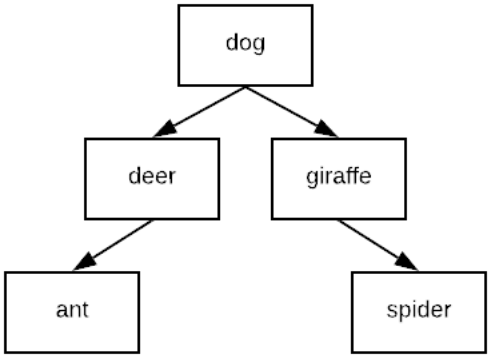
To compare how these two data structures perform, we'll start with data and analyze the operations needed to add the data into each data structure. Your comparison will involve reading text from a file and inserting the words into the linked list and the BST. Words need to be stored alphabetically, which means that your search function needs to find the correct insert location for each word, and return the number of node comparisons in the search process. Once the insert location is found, the insert operation is a constant and you don't need to count those steps.

For example, imagine you have the following linked list, and you want to search for the word "elephant".



You would start at the head of the list, the "ant" node, and compare "elephant" to "ant", "deer", "dog", and "giraffe", for a total of 4 comparisons. Once you reached "giraffe", you would know that "elephant" isn't in the list and it would be added after "dog". The cost for the insert operation would be the search cost, we're not counting the cost to update the pointers to link "elephant" into the list.

Assume you have a BST with the same five nodes that are stored in this order:



If you search for the word "elephant", you would compare "elephant" to "dog" at the root, then "giraffe". The left child of "giraffe" is where "elephant" would be found if it were in the tree, which happens after 3 comparisons if "elephant" is in the tree and 2 comparisons if it isn't.

Test Cases and Text Analysis

Using the *Hemingway_edit* file provided, generate a test case using the first 200 lines in the file, where you add each word to the linked list and the BST data structure. For each word read from the file, the word is either added to the data structure, or the count for the word is updated, similar to how you updated the word counts in Assignment 3. Each time the data structure is searched, output the number of comparisons for that word to either add it to the data structure or update the count for the word.

Calculate the average search operations and the standard deviation for the linked list and BST for an input file with 200 lines. Next, repeat the test for a larger and larger input file size, increasing the number of lines used by 200 with each test. You should have a total of 8 tests for each data structure.

When you run your program, you can direct the output to a file using

./Assignment6 <input file> >> outputFile

Where *outputFile* is the name of the file where the program output will be stored. If you run your program this way, nothing will print on the screen, instead, the output will be written to a file. This will make it easier to collect data.

Using the data collected from running your program, produce a graph that shows the number of lines in the file on the x-axis and the number of search operations on the y-axis. You can produce a graph for the linked list and BST separately, or place both curves on the same graph. Submit your graph with your code as an image.

After reading the file and building the data structures, output the number of unique words stored in the linked list and the BST. They should be the same. Next, output the total word count from each data structure, which is the unique words * the number of times the word was encountered. They should be the same.

Printing the Tree

After your code builds the data structures and produces the output for the text analysis, you should display a menu with four options:

Print Word - If the user selects *printWord*, then your program should ask for a word. If the word is in the tree, print the number of times it was found in the text, as well as the parent word, and left and right child words. You don't need to print any information in the linked list.

Print Tree In Order - If the user selects *printTreeInOrder*, do an in-order traversal of the tree and print all of the words. You don't need to print any information in the linked list.

Find Words in Range – If the user selects this option, they should be prompted for two words, and your program should print all words in the tree that are between the two words. The second word should follow the first word alphabetically, and if either word doesn't exist in the tree, you can print an error message.

quit – If the user selects quit, the program should quit.

The menu options are as much a debugging tool as anything. Use the print functions to check that you tree is built correctly.

Here are the prototypes for the methods included in the BST.h header file.

Public:
void BST::printWord(std::string word)
*/*Print the word, the count for the word, and the parent, leftChild, and rightChild words. Output format should be:*
Word: <word>
Count: <count>
Parent: <parent word>
Left Child: <left child word>
Right Child: <right child word>
**/*

void BST::printInOrderBST()
*/*Prints the nodes in the BST using an in-order traversal. */*

int BST::countBSTNodes()
*/*Counts the total number of nodes in the tree and returns that number. */*

int BST::addWordNode(std::string word)
*/*Add a new node to the tree. If node already exists in the tree, update the count for the node. Function returns the number of comparisons in the search. */*

void BST::findAlphaRange(std::string word1, std::string word2)
*/*Find and print all the words in the tree between word1 and word2 in alphabetical order. If either word1 or word2 don't exist in the tree, print an error message, such as "one or more words doesn't exist in tree".*

Private:
*void BST::printBST(BSTNode * node)*
*/*Method called in the printInOrderBST, starting at the root.*/*

*void BST::printPreOrderBST(BSTNode * node)*
*/*Method called in the public printPreOrderBST, starting at the root.*/*

*void BST::countBSTNodes(BSTNode *node, int *c)*
*/*Method called in the public countBSTNodes, starting at the root. Takes an int pointer that is updated with node count.*/*

BSTNode BST::searchBST(BSTNode *node, std::string word, int &opCounter)*
*/*Method called in addWordNode or findAlphaRange to determine if a node is in the tree. Update opCounter with the number of comparisons in the search process.*/*

Suggestions for completing this assignment
There are several components to this assignment that can be treated independently. My advice is to tackle these components one by one, starting with converting your assignment 4 code to work for a generic linked list of words and their counts in the file. Next, build the BST.cpp file to add words to the tree. Use simple cases rather than reading the words from the file. Once you have both the linked list and BST working, print both structures to verify that the words are stored correctly. Next, read from a small test file and verify that the words are added correctly and the counts are updated when words are encountered multiple times. You should be confident that your BST and linked list are working correctly before you run your tests counting search operations with multiple file sizes.

There are several examples of how to work with BSTs in Chapter 9 in your book, and we will also be covering these concepts in lectures this week and next week.

What to Submit
Your code should be separated into five files - *BST.h, BST.cpp*, *LinkedList.h*, *LinkedList.cpp* and *Assignment6.cpp*. You can compile your code on the command-line using g++

g++ -std=c++11 BST.cpp LinkedList.cpp Assignment6.cpp -o Assignment6

and then run your program on the command-line using

./Assignment6 <input file>

Your submission also needs to include the graph you produced showing the results of your comparison between the linked list and BST.

What to do if you have questions
There are several ways to get help on assignments in 2275, and depending on your question, some sources are better than others. There is a discussion forum on Slack that is a good place to post technical questions, such as how to insert a node in a linked list. When you answer other students' questions on the forum, please do not post entire assignment solutions. The TAs are also a good source of technical information, especially questions about C++. If, after reading the assignment write-up, you need clarification on what you're being asked to do in the assignment, the TAs and the Instructor are the best sources of information.