Project 3 - Hashed Word Frequency Splay Tree

Due: Please check due date on BlackBoard

Objectives

The objective of this programming assignment is to have you practice using recursion and iteration in your programs and to familiarize you with the Splay tree data structure.

Introduction

In Grand Theft Auto, you can play several players. Each player has a cell phone. Much like our own cell phones, words we routinely use should show up after we type in a few letters. It would be nice if that feature would have been included in the cell phones. While the application you are about to create does not do that, it will at least get it into the right direction.

Application Setup

Using a loosely hashed **Splay Tree** and a file with sample text, you will organize a tree with the data given. The hashed portion with contain 26 individual Splay Trees, for each of the letters. ‘A’ and ‘a’ will be stored in index 0,’B’ and ‘b’ in index 1, and so forth. The order of whatever text in the file read in will determine the structure of the tree. Below, is an incomplete example of the words starting with T’s read from the provided sample file.

|  |  |
| --- | --- |
|  | Please note that the tree does not contain the frequencies that each node should have and this tree would be incomplete with the same file provided. Details about frequency are below.  The **infix** order of this example tree would be:  Tags  Tested  This  tag  tags  than  the  this  to |

Since the text many contain the same word several times, the tree will also contain how many times the word appears, it will NOT contain the same word in multiple nodes and case must be preserved. Any methods added to the Splay Tree will need to be generic. The features we will ask you to create will be used test how efficient your tree is.

Hashed (kinda) Table Data Structure

The base data structure that will hold all of the Splay Trees will be an **Array** called “table”. The size of the table will be 26 (one for each letter A/a, B/b, etc…) for our application, but notice you should set the length from whatever is creating in instance of the array. (In our example, Driver.cpp will be creating the instance of the HashedSplay called wordFrequency.)

Within each index of the array, a Splay Tree will be created. That lone Splay Tree will contain only the words that start with a particular letter. Again, all words starting with “A” or “a” will go into the 0 index.

Generic Splay Tree Data Structure

The Splay Tree data structure called SplayTree<AnyType> that will hold all nodes that will contain the word and a frequency that contain the same letter. Our Splay Tree must use recursion and iteration to complete every operation described in the assignment below.

There will be SOME instances where the generic nature of the Splay Tree will need to be bypassed by casting the AnyType x to a Node. The one tricky issue we found was how to increment the node’s value.

Print HashTable Results Application

This will be used as a grading test. Using the entire array “table” and named **printHashCountResults()**, this will print the root value (to the screen) at each index in order and display how many nodes each tree contains. The results will look like below:

This tree starts with Node [word=a, frequency=7] and has 22 nodes

This tree starts with Node [word=be, frequency=7] and has 12 nodes

This tree starts with Node [word=Contributors, frequency=2] and has 18 nodes

This tree starts with Node [word=Donate, frequency=1] and has 16 nodes

This tree starts with Node [word=eliminate, frequency=1] and has 8 nodes

This tree starts with Node [word=following, frequency=1] and has 14 nodes

This tree starts with Node [word=GPLv, frequency=1] and has 4 nodes

This tree starts with Node [word=httpexamplecom, frequency=1] and has 11 nodes

This tree starts with Node [word=is, frequency=19] and has 16 nodes

This tree starts with Node [word=just, frequency=1] and has 3 nodes

This tree starts has no nodes

This tree starts with Node [word=list, frequency=5] and has 12 nodes

Please note your output may differ in detail.

Print Tree in Infix Notation Application

This method named **printTree( )** will print the Splay Tree in infix order. Notice that a single tree (index) will be selected. Notice that in the case below, the “T/t” or index 19 of “table” was selected.

You should be able to print the tree by passing either the index number OR the letter you wish to print. At the end of the input you will also print the number of splays the tree has seen.

Consult the example Driver to see how the call for printTree() was done. The results should look like this:

Node [word=Tags, frequency=2]

Node [word=Tested, frequency=2]

Node [word=This, frequency=7]

Node [word=Thus, frequency=1]

Node [word=Titles, frequency=1]

Node [word=tag, frequency=8]

Node [word=tags, frequency=2]

Node [word=tagsreadmetxt, frequency=1]

Node [word=tagsscreenshotpng, frequency=1]

Node [word=take, frequency=1]

Node [word=taken, frequency=1]

Node [word=templates, frequency=1]

…

This tree has had 164 splays.

Please note your output may differ in detail.

FindAll/Partial Application

After the tree has been filled with data, this application, much like texting will find words within a certain tree index. Using the tree pictured above, using “th” would give the results “the”, “this” and “than” **in infix order**. The output of this function should look like below:

Printing the results of nodes that start with 'the'

Node [word=the, frequency=49]

Node [word=then, frequency=1]

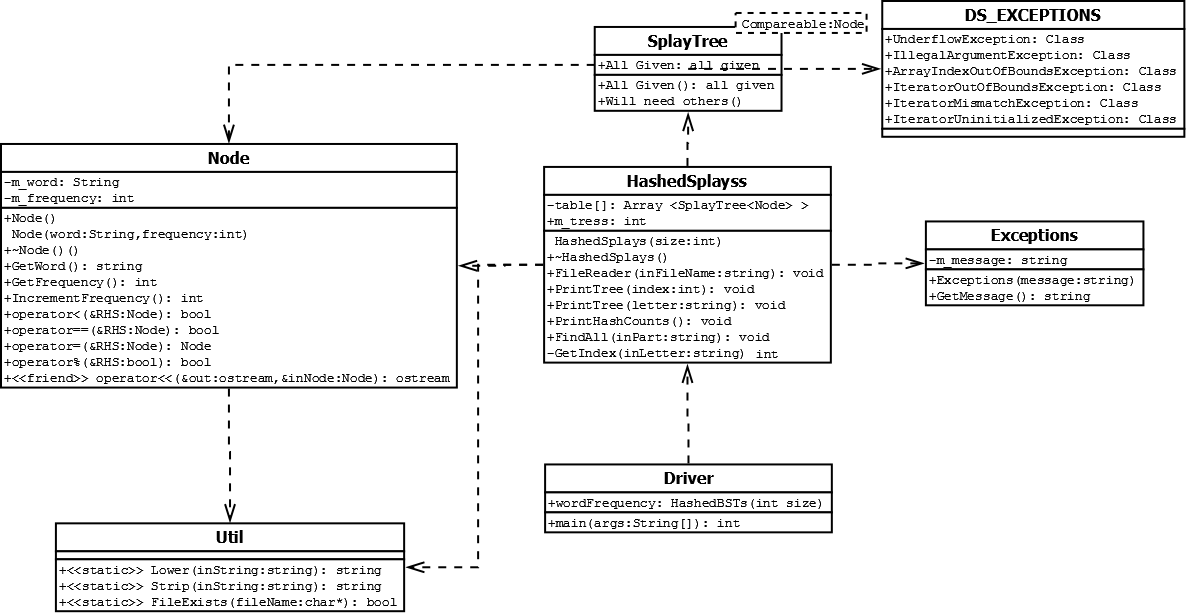
**Note: This function does not splay at all!! Think of it as a debugging tool.**

Coding and Implementation Requirements

*Note: Running time is one of the most important considerations in the implementation of a data structure. Programs that produce the desired output but exceed the required running times are considered wrong implementations and will receive substantial deductions during grading.*

1. Because we are using splay trees, it is mandatory that any operation which traverses the tree (except FindAll and PrintTree) must perform a splay at the last traversed node.
   1. This includes any function that you may choose to add to the given splay tree code.
   2. That said, you should be efficient in your code, which would be reflected by judicious calls to the internal splay function.
   3. Grading will therefore have a small tolerance in splay count, but marks will be deducted for efficiency if there are excessive splays.
      1. excessive splays USUALLY come from using contains, THEN inserting (so it almost doubles). Just insert, and if it finds a duplicate, increase the frequency, then splay
2. You may use previous files from Project 2 to help you. (Exceptions, deExceptions, etc…)
3. You should also use the **same** filter as you did in Project 2.
   1. if there is a discrepancy in this document, inform the coordinator and ignore the error.
4. You may use C++’s Array or Vector class to hold the generic Splay Tree. It is to be named *table*.
5. You will be required to use a generic Splay Tree. You may start with the Splay Tree class from the textbook or design your own. Each option has advantages and disadvantages, **like you won’t have to change much of the code in the generic Splay Tree given**, but you might not understand what all is going on in the code. So, one component of grading will evaluate how elegantly you employ iteration or recursion to implement this data structure. (Yes, you are being graded on aesthetics!) You are allowed to add functionality to the given code as well.
6. Words given in a file will ultimately be stored in a Splay Tree. But the file will have other data other than words. You will need to filter this. Filtration is specified below. (Just make sure there are still spaces!)
7. The run time for the entire project should not be anything more than O(n), BUT:
   1. all Splay Tree operations will run in O(log n) time
   2. all HashedSplays will operate in O(n) time
8. The main base class will be named HashedSplays. It **must** contain:
   1. **class** HashedSplays.
   2. constructor that will accept the size of the hashed table
   3. the Array *table*
   4. the function printHashCountResults()
      1. this is a grading and debugging tool
      2. should run in O(n) time
   5. the function fileReader(String filename)
      1. this function will open the file, filter it to the distinct words, remove ALL punctuation, (except – and ‘ inside words) **and numbers**, then places those words into the appropriate Splay Tree in the Array *table*.
      2. should run in O(n) time
   6. the function findAll(String sample)
      1. this function will collect all codes that START with the letters in the sample Node and print that list of nodes
      2. this will run in O(log n) time
      3. **this function does not splay. This is a debugging tool.**
9. The class Node will be given to you to save some time. Look carefully how items in *Node* are to be accessed. **This class/file will need major work.**
   1. [Node.](http://userpages.umbc.edu/~slupoli/notes/DataStructures/projects/WordFrequencyHashedTreesS14/code/Node.java)cpp is a very basic class that holds an instance of a word and it’s frequency.
   2. The Node’s frequency will be updated if another word of the same name is found in the file.
   3. ***There will be OPTINAL and MINIMAL alterations to the Node.cpp/.h file***
10. The class SplayTree will be the heart of your program.
    1. Must be declared **class** SplayTree<AnyType>
    2. Recursion **must** be present in any traveling within the tree for displaying nodes, inserting, searching, etc…
    3. Some of the required functions include:
       1. insert (boot strap and iterative version)
       2. printRoot (boot strap and return function)
       3. printTree (boot strap and recursive version)
       4. please use the UML below for further details (**and about boot strap**)
    4. You will be required to keep track the number of splays chains the tree has seen. This means every time a splay starts you need to count it. You do not need to keep track of the number of rotations. For example if there is a request to find a node this should trigger a splay in the tree and you need to count that as 1, even if it takes 50 rotations to move the node to the root.
    5. make sure Splay is ONLY when inserting or searching (contains)
    6. the rest of the support functions are up to you
11. **There will be SOME instances where the generic nature of the Splay Tree will need to be bypassed by casting the AnyType x to a Node. The one tricky issue we found was how to increment the node’s value. These are the only cases!!**
12. Check that your code compiles correctly with this sample **main** program: [Driver.cpp](http://userpages.umbc.edu/~slupoli/notes/DataStructures/projects/WordFrequencyHashedTreesS14/code/Driver.java) without alteration.

A UML diagram below may help you in the right direction.



**You *can* (and should) create more methods than what the UML Diagram shows.**

**Just watch for the requirements.**

## What to Submit

Read the [course project submission procedures](http://www.csee.umbc.edu/courses/undergraduate/341/fall13/projects/submission.shtml). *Submission closes by script immediately after midnight.* Submit well before the 9:00pm deadline, because 9:00:01 might already be late (the script takes only a few seconds to run).

You should copy over all of your C++ source code and have your .cpp files in their own directories which are in turn under the src directory. You must also supply an MAKE build file.

Make sure that your code is in the ~/cs341proj/proj3/src directory and not in any other subdirectory of ~/cs341proj/proj3/. In particular, the following Unix commands should work.

cd ~/cs341proj/proj3/src

make

make run **DATA=**data.txt (data.txt could be ANY filename)

make clean

Addendum

None yet!