Project 2, Hashed Word Frequency BST

Due: Please check due date on BlackBoard

Objectives

The objective of this programming assignment is to have you practice using recursion in your programs and to familiarize you with the binary search 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 **generic** **unbalanced** Binary Search Tree and a file with sample text, you will organize a tree with the data given. The hashed portion with contain 26 individual BST, 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 sample file provided.

|  |  |
| --- | --- |
|  | 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. We are asking you to make it generic since we may test it with other data types. There will be a few features we will ask you to create in order to test how efficient your tree is.

Hashed (kinda) Table Data Structure

The base data structure that will hold all of the BST will be an ArrayList called “table”. The size of the table will be 26 (one for each letter A/a, B/b, etc…) for our application, but notice that I want to be able to set the length from whatever is creating in instance of our ArrayList. (In our example, Driver.java will be creating the instance of the HashedBST.)

Within each index of the ArrayList, a BST will be created. That lone BST will contain only on type of words that start with a letter, ignoring case.

Generic Binary Search Tree Data Structure

The BST data structure called BinarySearchTree<AnyType **extends** Comparable<? **super** AnyType>> that will hold all nodes that will contain the word and a frequency that contain the same letter. Our BST must use recursion to complete every operation described in the assignment below.

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 ArrayList “table” and named printHashCountResults(), this will print the root value at each index in order and display how many nodes and the first nodes each contain. 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

Print Tree in Infix Notation Application

This method named printTree( ) will print the BST 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. Consult the example Driver to see how this 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]

…

Lowest Common Ancestor Application

After the tree has been filled with data, we will find the lowest common ancestor. Using the tree example above, and given two values, (let’s say “This” and “this”) it will print the lowest common ancestor, (which would be “tag” in the example above). Named findCommonAncestor(AnyType x, AnyType y), it will display the results. You will need supporting functions to complete this. You will also be graded on HOW efficient your application runs and what resources it requires. The overall concept further described here: <http://en.wikipedia.org/wiki/Lowest_common_ancestor> .

Node [word=This, frequency=1] and Node [word=than, frequency=1]

is: Node [word=tag, frequency=8]

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”. The output of this function should look like below:

Printing the results of nodes that startWith: 'the'

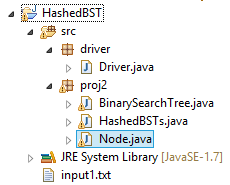
Node [word=the, frequency=49]

Node [word=then, frequency=1]

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.*

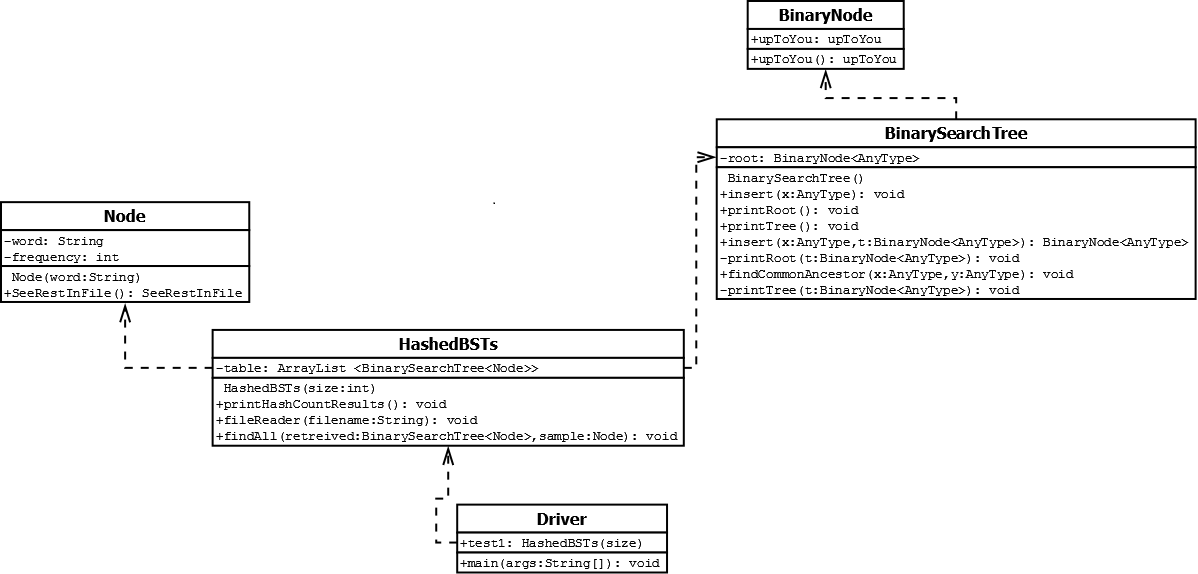
1. You will be required to use Java’s ArrayList to hold the generic BST. It is to be named *table*.
2. You will be required to develop a generic BST. You may start with the BST class from the textbook or design your own. Each option has advantages and disadvantages. A primary objective of this programming assignment is to have you use recursion. So, one component of grading will evaluate how elegantly you employ recursion to implement this data structure. (Yes, you are being graded on aesthetics!)
3. Words given in a file will ultimately be stored in a BST. 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!)
4. The run time for the entire project should not be anything more than O(n), BUT:
   1. all BST operations will run in O(n log n) time
   2. all HashedBST will operate in O(n) time
5. The main base class will be named HashedBST. It **must** contain:
   1. **public** **class** HashedBSTs <AnyType>
   2. constructor that will accept the size of the hashed table
   3. the ArrayList *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, (even the ‘ and - inside of words) **and numbers**, then places those words into the appropriate BST in the ArrayList *table*.
      2. should run in O(n) time
   6. the function findAll(BinarySearchTree<Node> retrieved, Node 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
6. The class Node will be given to you to save some time. Look carefully how items in *Node* are to be accessed.
   1. [Node.java](http://userpages.umbc.edu/~slupoli/notes/DataStructures/projects/WordFrequencyHashedTreesS14/code/Node.java) 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. The frequency will be updated ONLY IN THE BST class!
   4. There will be NO alteration to the Node.java file
7. The class BinarySearchTree will be the heart of your program.
   1. Must be declared **public** **class** BinarySearchTree<AnyType **extends** Comparable<? **super** 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 recursive version)
      2. printRoot (boot strap and recursive version)
      3. printTree (boot strap and recursive version)
      4. please use the UML below for further details (**and about boot strap**)
   4. the required lowest common ancestor application function must follow:
      1. **void** findCommonAncestor(AnyType x, AnyType y)
      2. you will be graded on HOW efficient your application runs and what resources it requires.
   5. the rest of the support functions are up to you
8. Your hashed BST class called HashedBST and must be accessible from a main program in a different package. You can check that your code compiles correctly with this sample **main** (has the main() ) program: [Driver.java](http://userpages.umbc.edu/~slupoli/notes/DataStructures/projects/WordFrequencyHashedTreesS14/code/Driver.java). This test program must be placed in a separate directory named Proj2 (since it belongs to the Proj2 package). Your code must compile with Driver.java without alteration.



**Notice how the input files are placed inside of the project. When we test, we will gather input from that location.**

1. In a new twist, we will be looking for custom APIs. Some of your commenting is… suspect at best. For the instructional staff, your custom APIs will greatly help us help you. For a video on how to create Custom APIs, watch [this](http://userpages.umbc.edu/~slupoli/notes/Java/videos/Eclipse/Creating%20your%20own%20APIs.html). For having ANT build them for you, well, do some research.

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 11:59pm deadline, because 12:00:01 might already be late (the script takes only a few seconds to run).

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

Make sure that your code is in the ~/cs341proj/proj2/ directory and not in a subdirectory of ~/cs341proj/proj2/. In particular, the following Unix commands should work.

cd ~/cs341proj/proj2

ant compile

ant run

ant clean

Addendum

None yet!