Project 3, AutoFill using RB Trees and Heaps

Due: Please check due date on BlackBoard

Objectives

The objective of this programming assignment is to have you practice using Red-Black trees (known as RBT from here on) to quicken a search and prioritize values using a max heap.

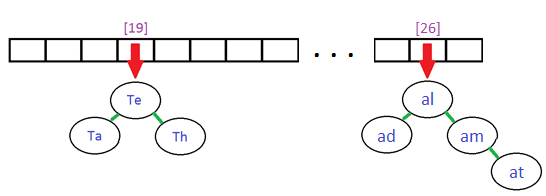
Introduction

While you are texting in class, words we routinely use should show up after we type in a few letters. This feature is to help us text faster. (Even though talking is still faster!!) Frankly, it’s a little scary that after a very large sampling, (and proper English wouldn’t hurt), the autofill will be correct in many cases.

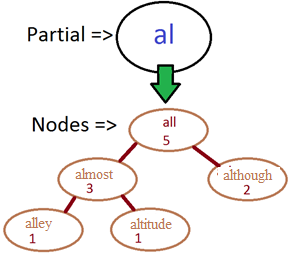
Application Setup

We will be **re-**using a loosely hashed **generic** RBT with a nested max heaps inside of each node and a file with sample text.

Starting with the hashed RBTs, will be organized by the FIRST letter in the hashed table, but they will only contain the first two letters (or less) of the words read in from the file in the RBT. The first two will be used to help us autofill the word we want to type.



Each node in the RBT is called “Partial” which again will contain the maximum of two letters. But within each “Partial” node, contains a max heap called matches that will contain “Nodes” (same Node we have been using) of the REAL complete words and their frequencies. You are given the code to “Partial”. Take a close look.



You will organize the RBT and max heaps with the data file given, which is in the form:

Node [word=A, frequency=32]

Node [word=About, frequency=2]

Node [word=Adam, frequency=3]

We are asking you to make every class 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 application is.

Hashed (kinda) Table Data Structure

The base data structure that will hold all of the RBTs will be the same ArrayList called “table”. The size of the table will be 52 (one for each letter A, B, …, a, 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 HashedRBTs.)

Within each index of the ArrayList, a RBT will be created. That lone RBT will contain only on type of **partial** words that start with the first two letters, ***NOT*** ignoring case.

Generic Red Black Tree Data Structure

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

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 Partial value ONLY at each index in order and the heaps values that are contained in that Partial. Notice this is only the ROOT node of the RBT, since this could be a massive display if printing every node in the RBT. The results will look like below:

This tree starts with Partial [word=we] --> The heaps contains:

[1] Node [word=weather, frequency=8]

[2] Node [word=well, frequency=4]

[3] Node [word=wellbred, frequency=3]

...

This tree starts has no nodes

This tree starts with Partial [word=ya] --> The heaps contains:

[1] Node [word=yacht, frequency=4]

[2] Node [word=yachts, frequency=2]

[3] Node [word=yachtsman, frequency=1]

[4] Node [word=yachtunless, frequency=1]

Print Red-Black Tree in Infix Notation Application

This method named printTree( ) will print the RBT in infix order, and should again print the heap within each Partial. Notice that a single tree (index) will be selected and all of the answers are “Partials”. Notice that in the case below, the “A” or index 0 of “table” was selected. Consult the example Driver to see how this was done. The results should look like this:

Partial [word=A] --> The heaps contains:

[1] Node [word=A, frequency=3]

Partial [word=Ab] --> The heaps contains:

[1] Node [word=About, frequency=2]

[2] Node [word=Aboot, frequency=1]

Partial [word=Ad] --> The heaps contains:

[1] Node [word=Adam, frequency=3]

[2] Node [word=Admiral, frequency=1]

Partial [word=Af] --> The heaps contains:

[1] Node [word=African, frequency=8]

[2] Node [word=After, frequency=7]

[3] Node [word=Afropath, frequency=1]

[4] Node [word=Africa, frequency=2]

Partial [word=Ag] --> The heaps contains:

[1] Node [word=Again, frequency=6]

[2] Node [word=Age, frequency=1]

Partial [word=Ah] --> The heaps contains:

[1] Node [word=Ah, frequency=1]

Partial [word=Al] --> The heaps contains:

[1] Node [word=All, frequency=8]

[2] Node [word=Almanach, frequency=3]

[3] Node [word=Alamance, frequency=2]

[4] Node [word=Although, frequency=2]

[5] Node [word=Allow, frequency=1]

[6] Node [word=Almost, frequency=1]

[7] Node [word=Also, frequency=1]

[8] Node [word=Alas, frequency=1]

…

Print Immediate Matches Application

This application is the closest thing to out cell phone text matching. After all data has been processed, Given a Partial, the function should return the highest 3 values in the max heap of that partial. Please look at the driver to determine how this is called. The result(s) should look similar to this when there is and is not a match:

\*\*\* Printing immediate matches for 'Ar'

Node [word=Are, frequency=6]

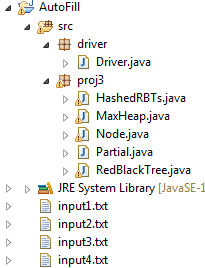
Node [word=Army, frequency=2]

Node [word=Arent, 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 RBT. It is to be named *table*.
2. You will be required to develop a generic RBT. You may start with the RBT 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 and frequency given in a file will ultimately be stored in a RBT’s max heap. But the file will have other data other than words. You will need to filter this. What to filter will be apparent when you look at the example input files.
4. The run time for the entire project should not be anything more than O(n), BUT:
   1. all RBT operations will run in O(n log n) time
   2. all Max Heap inserting operations will run O(log n)
   3. printing in the entire Max Heap operations will run O(n)
   4. printing in the top 3 maximum values and root Max Heap operations will run O(1)
   5. all HashedRBTs will operate in O(n) time
5. The main base class will be named HashedRBTs. It is VERY similar to HashedBST. It **must** contain:
   1. **public** **class** HashedRBTs
   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 the useless text it to the distinct words, and frequency, then places those words into the appropriate RBT in the ArrayList *table*, then into the appropriate Partial, then into a max heap.
      2. should run in O(n) time
   6. the function retreiveHashedRBTat(int index)
      1. returns the RBT at that index
      2. should run in O(1) time
6. The class RedBlackTree will be on your own, but must contain:
   1. **public** **class** RedBlackTree<AnyType **extends** Comparable<? **super** AnyType>>
   2. must be entirely generic, there is no need for “instanceOf” like BinarySearchTree
   3. Recursion **must** be present in any traveling within the tree for displaying nodes, inserting, searching, etc…
   4. constructor, compare, insert, contains, which are standard RBT functions
   5. also needs functions to reorient and rotate if one side is overloaded
   6. the function printTree()
      1. bootstrap and recursive function
      2. recursive function prints the tree in infix
   7. the function printRoot()
   8. the rest of the support functions are up to you
7. The class MaxHeap will be on your own, but must contain:
   1. **public** **class** MaxHeap<AnyType **extends** Comparable<? **super** AnyType>>
   2. standard Heap functions (can find more details in your book/notes)
   3. the basic but custom functions:
      1. printHeap() which runs in O(n) time
      2. toString()which runs in O(n) time
      3. printHeapRoot() which runs in O(1) time
   4. the function printImmediateOptions()
      1. this function will collect the first three values in the MAX heap
      2. this will run in O(1) time
8. The class Node will be given to you to save some time. ***It has changed slightly.*** Look carefully how items in *Node* are to be accessed.
   1. [Node.java](http://userpages.umbc.edu/~slupoli/notes/DataStructures/projects/AutoFillHeapsS14/code/Node.java) is a very basic class that holds an instance of a word and it’s frequency.
   2. The Node’s frequency and word is given IMMEDIATETLY in the file.
   3. There will be NO alteration to the Node.java file
9. The class Partial will also be given to you to save some time. Look carefully how items in *Partial* are to be accessed.
   1. [Partial.java](http://userpages.umbc.edu/~slupoli/notes/DataStructures/projects/AutoFillHeapsS14/code/Partial.java) holds an instance of a word first two (or less) letters. No frequency!!
   2. But Partial also contains the MAX HEAP instance. This will store all of the COMPLETE words that start with Partial.
   3. You are still required to create the heap class.
   4. There will be NO alteration to the Partial.java file
10. Your hashed RBT class called HashedRBTs 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/AutoFillHeapsS14/code/Driver.java). This test program must be placed in a separate directory named proj3 (since it belongs to the proj3 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. We will be looking for custom APIs. This will be a part of your grade. You are to create an API comment for every function YOU create.

No UML will be given for this project.

## 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/proj3/ directory and not in a subdirectory of ~/cs341proj/proj3/. In particular, the following Unix commands should work.

cd ~/cs341proj/proj3

ant compile

ant run

ant clean

Addendum

None yet!