CS 2420: Introduction to Algorithms and Data Structures

Assignment 07 – Spell Checker

Fall, 2019

Objectives

The purpose of this assignment is to implement a Binary Search Tree (BST) of Strings and use it as an underlying representation for a low-level spell checker.
Note: this is a pair programming assignment. You **must** work with a partner for this assignment. And, if you haven't changed a partner yet in this semester, you should work with a NEW partner. (Indicate whether you are working with a new partner or not in the analysis document.)
This assignment hits these learning outcomes stated in the syllabus:

LO1 implement, and analyze for efficiency, fundamental data structures (including lists, graphs, and trees) and algorithms (including searching, sorting, and hashing)

LO2 employ Big-O notation to describe and compare the asymptotic complexity of algorithms, as well as perform empirical studies to validate hypotheses about running time

LO4 apply algorithmic solutions to real-world data

LO6 appreciate the collaborative nature of computer science by discussing the benefits of pair programming

Important

- You are supposed to submit, in addition to your .java source, .java JUnit test files (both mandatory), a design document (optional) and an analysis document (mandatory).
- The design document should only be written before you start writing code.
- The analysis document should only be written after you have finished programming and have thoroughly tested your code
- So make sure to leave plenty of time to work on these documents as well as your code.

Setup

1. If you haven't already, please follow the instructions here and get your Eclipse environment set up the way we want for this course.
2. Make a copy of the *entire* `ProjectTemplate` eclipse project in the same workspace. (Right click `ProjectTemplate` > `Copy`; then right-click anywhere in the project explorer, and click `Paste`.) Name it `Assignment07-SpellChecker`. Delete `HelloWorld.java` in the Project Explorer.

3. Download a new version of the `components.jar` from Canvas > Files > components (link).

4. Download the associated files from this Canvas folder (link). Place the source files in the proper `src/` folder under your project, and JUnit files under a `test/` folder (create one if required), and the `.txt` files in the `data/` folder (create one if required).

5. Make sure, each file has `package assignment07;` at the top.

Background

- We discussed both binary search trees (BSTs) in class. Their key property is, for a non-empty BST, every node in the left subtree is "smaller than" the root, and root is "smaller than" every node in the root. And, of course, this relation holds for every subtree.

- All major editing programs (from Eclipse to MS Word) incorporate spell checking. This project will show how to create a "word verifier" which is a low-end spell checker. The *dictionary* will be specified as a sorted set and will use a Binary Search Tree implementation.

- As always, you are required to create this code from scratch and are **not** allowed to use pre-written code. You are of course allowed to view the online notes/readings/videos and discuss general ideas with your classmates, but should use these to guide your learning, not to provide you with boilerplate code.

- We know that the key to an efficient spell-checker is a *fast* mechanism for storing and searching the set of valid words. It should be noted, that the underlying implementation of this set will not be a singly linked list nor even an array, but instead will be a binary search tree (BST).

- To keep things simple, you will implement a BST of Strings (not generic types). The implementation should include a class to represent a binary tree node (very to similar to how linked list code has a nested Node class). Please note that there is **no requirement that the BST be balanced**.

- Once the BST is completed, you will apply it to the problem of spell-checking a document. Thus you will implement a class to represent a spell-checker and a main program to demonstrate the spell-checker for various valid-word-sets and document files.

Informative Pictures showing Operations

- For both the insert node and delete node methods on the Binary Search Tree you should draw images to explain what is happening. In both cases you need to show the "traversal" to the proper node and then the "action" of removing/adding.

- You are encouraged to use `https://www.draw.io` to complete your pictures, but are welcome to use other drawing tools that allow you to export (for your PDF analysis document).

- It is strongly recommended that for every method you write, you draw out a picture (perhaps not as formal as the above) and create "paper-ware" showing the proper way to assign the references in the node.

  – WARNING: The TAs will be told not to help you if you cannot show your pictures for the method you are writing.

Method

Design Document: not required, but highly recommended

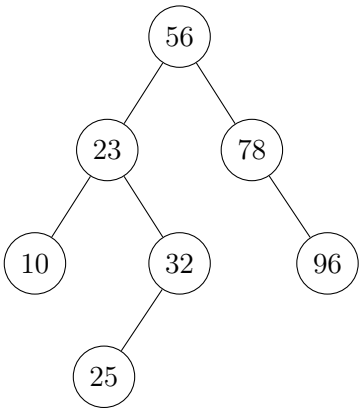
- Must have both partners' names at the top and it must be named "spellcheckerDesign.pdf"
- The purpose of this document is to show your prior conceptions (misconceptions?) about how the program will be designed and what the likely outcome of running the program will be.
- There are a lot of interesting issues in this assignment and you should discuss and resolve those using pen-and-paper instead of code with your partner *before* you start coding.
- You are encouraged to include a lot of diagrams of various operations on your BST in this document.

Implementation

Underlying BST

Implement all the methods in `BinarySearchTreeOfStrings.java` as discussed in class. Make sure you understand the method contracts in the given file. Some illustrations below should help in this implementation.

- To implement this class, first, create an internal `Node` class that lets you represent a BST node. Feel free to add any other fields you may need.
- Then, use the algorithms discussed in class. Feel free to create any number of private helper methods to complete this implementation.
- You should also override the `toString()` method for this class, and have it output the nodes in the BST using **In-Order traversal**, enclosed in `[` and `]` (See an example below). This will be used for testing purposes.



return value of `toString()` on the BST above is `[10,23,25,32,56,78,96]`.

- You are not required to implement an iterator or the `equals` method on the BST.

SpellChecker.java

Take a look at the given `SpellChecker.java` interface. You will implement this interface by providing a class that has a `BinarySearchTreeOfStrings` as the underlying representation.

- loadValidWords: This method takes a file containing one word per line and loads them on the underlying tree.

```
/**
 * Loads the list of valid words from the given file.
 *
 * @param filename relative or absolute path to the file from which to lead
 *                 valid words
 *
 * @requires file contains one word per line, i.e., anything that appears on a
 *           line, a group of letters, numbers, special characters, etc. will be
 *           considered a valid word.
 */
void loadValidWords(String filename);
```

- misspelledWords: This method opens an input text file and returns a list of words that are considered "misspelled" according to the underlying set of valid words.

```
/**
 * Returns a list of misspelled words in the given file, based on the valid
 * words for this instance of the spell checker.
 *
 * @param filename relative or absolute path to the file to be spell checked
 * @return list of invalid words
 */
List<String> misspelledWords(String filename);
```

- clear: self explanatory; this is needed so we can check various valid-word-sets v. various input files combinations for our spell checker.

```
/**
 * Clears this instance of the spell checker, i.e., resets the valid words set
 * to empty.
 */
void clear();
```

Testing

- A thorough test plan is provided to you already (35 test cases). Those are the tests the autograder will run. Your code must pass all the test cases to receive full credit. So, instead of relying on the autograder, you should make sure your runs of JUnit in your eclipse are problem-free.
- You are not required to turn in the JUnit files for this assignment.

Documentation

- You should not need to create any public methods, but you may want to create some private helper methods. Make sure you write proper Javadoc comments and tags on all such methods you add. Make sure you update all the `@author` tags with both your names as well.
- In-code comments are useful to help a reader (in our case, a TA) understand what you are trying to do. It goes without saying – making your TA's life easier is going to make your life easier (in terms of grades). However, do not "over-comment" your code. Lines that are straightforward and obvious do not need comments.

Analysis Document

- Must have both partners' names at the top and it must be named "spellcheckerAnalysis.pdf"
- This course is not only about creating software to do "something" but also about understanding why programs (algorithms and data structures) work in the way they do. When you are satisfied that your program is correct, you are to write an analysis document discussing your findings.

As part of preparing for your analysis:

Experimentation

Once everything is working and tested, write a separate main program in a file called `Timing.java` to experiment with some properties of BSTs. Here are two experiments to run:

1. Design and conduct an experiment to illustrate the effect of building an N-item BST by inserting N (unique) items into the BST in sorted order versus inserting the same N (unique) items in a random order. Carefully describe your experiment, so that anyone reading this document could replicate your results. You should time the insertion method to see how long it takes to build the tree.

    - Once the items are inserted, you should also time the contains method run over all of the items.
    - Due to the randomness of inserting in "random" order, you will want to perform your experiment several times (with a new random ordering each time) and record the average running time required.

2. Consider the `BalancedBST1` class in components.jar (API link), which re-balances itself when necessary.

    - Design and conduct an experiment to illustrate the differing performance for this "balancing" BST vs your non-balancing BST.
    - Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment.
    - Note: Again, *your* implementation is not required to balance itself.

Graphs/Plots

- The experiments above should produce timing tables which can be (and should be) plotted using Excel or other software.
- Make sure the figures built in your timing study are informative and easy to understand. Since the organization of your plot(s) is not specified here, the labels and titles of your plots(s), as well as, your interpretation of the plots is critical.
- If the y axes on your graphs are similar, you could combine two plots on one graph. Don't do this if by doing so, you distort or make your graphs harder to read.
- Note: Please make sure the code required to conduct your experiment and print out the data is in the Timing.java file and should be well-commented.
- Finally, the purpose of the graphs is to illuminate what is going on. If the graphs are not informative, you are not doing your job. Additionally, you will want prose in your analysis document referencing your graphs and explaining what they mean.

Handing in

This assignment is due Thursday, October 31, 11:59 pm MDT. Late penalty is as described in the syllabus.
Submit the .java files you modified/created (source and test files) and the .pdf file(s) –design (optional) and analysis documents– on Gradescope. Once again, make sure all the files have both partners' names on them and submit the files directly, not in a zip bundle.