COMP 3160 – Fall 2014
Project 1: Spell Checkers

Number of People: Up to 2. If you work with someone else, include both your name and your teammate's name on all source code files. Only one person needs to submit the final product to eCourseware, but include a note with the submission that lists both of your names.

Due: Thursday, Oct. 23 by 1:00 pm

Honors Grading: This project is worth up to 110/100 points for regular students, but only 100/100 for honors students. If an honors and regular student decide to work together, both agree to be graded on the stricter honors scale. The point breakdowns throughout the assignment are given in **(regular/honors)** format.

Submission: Zip all of your Java source files (you can zip the entire project folder if using an IDE) into a single file and upload it to the proper folder in the eCourseware dropbox at https://elearn.memphis.edu.

Coding Style: Use descriptive variable names. Use consistent indentation. Use standard Java naming conventions for **variableAndMethodNames**, **ClassNames**, **CONSTANT_NAMES**. Include a reasonable amount of comments.

In this project you will implement and explore the performance of different kinds of spell checkers. I have provided a text file containing about 110,000 English words on eCourseware for you to use. You can assume that all words contain only lower-case letters.

Approach 1: Binary Search Tree

One easy way of implementing a spell checker is to use a binary search tree. If you have a list of valid words, you can insert them into a BST. Checking whether a word is spelled correctly then simply involves searching the tree for that word. If the word is found, great – if not, it must be misspelled! Remember that finding items in a BST takes $O(\log n)$ time on average, where n is the number of elements in the tree. Thus, the expected time cost of using this spell checker depends on the number of words that it's storing.

1. **(21/19 pts)** Write a **BSTSpellChecker** class. This class should have an instance variable of type **BinarySearchTree<String>** to store its word list. You can use the **BinarySearchTree** code that we wrote in lecture. **BSTSpellChecker** should contain the following methods:

    a. *(2 pts)* **void add(String s)**
    Adds the specified word to the tree.

    b. *(2 pts)* **boolean contains(String s)**
    Returns whether the specified word is in the tree.

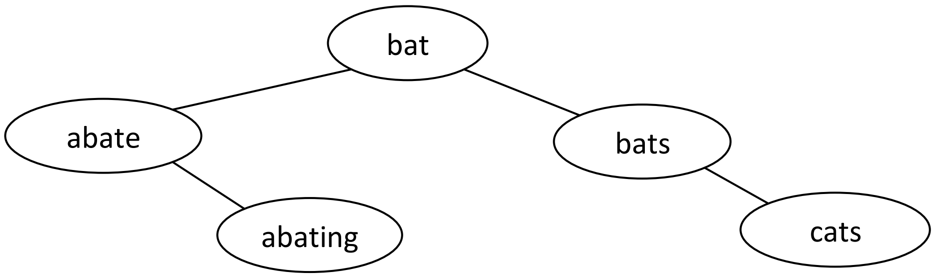
    c. *(6/5 pts)* A method that reads a list of words from a file and adds them all into the tree. Add the words to the tree in the same order that they appear in the file (i.e., alphabetically). (Note – you will probably have to use non-recursive versions of the BST **add** and **find** methods to prevent stack overflows here!)

    d. *(11/10 pts)* Remember that adding things in order into a BST is a bad idea – this will result in a linked list with $O(n)$ performance. Write another version of the above method that adds the words in a balanced manner. You can do this by first adding the word in the middle of the file, then recursively adding the words in the middle of the left and right halves, and so on.

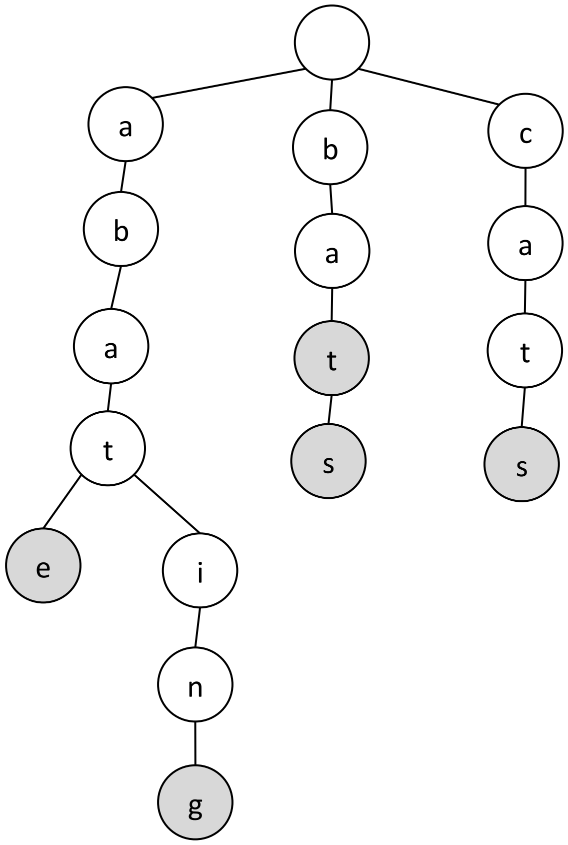
2. **(6/5 pts)** In a separate client program, create two new instances of **BSTSpellChecker**. One should read the words from file linearly; the other one should read the words in a balanced manner. Experiment with how long it takes to find words in both versions.

Approach 2: Trie

Although ensuring a balanced BST does wonders for the performance of your spell checker, there is another type of tree that is particularly well suited for this application. This type of tree is called a *trie* (often pronounced "try," even though the name was taken from the word re*trie*val). Tries are nice for storing words because they group together words with common prefixes. For example, suppose you want to store the words {"abate", "abating", "bat", "bats", and "cats"}. Using a BST (and adding in a balanced manner) would give you something like this:



Notice that many of the letters are repeated! By contrast, here's what a trie would look like:



The shaded nodes indicate points that form complete words. To check whether a word exists in the trie, start from the root and work your way down the trie one node at a time, according to the letters in the word. If you end on a shaded node, then it's a valid word. If you don't end on a shaded node (e.g., "abat" in the trie above), or if you're unable to find the next letter in the trie (e.g., "abalone" in the trie above), then the word is not in the trie.

Checking whether a word exists in a trie depends on the length of the word to be checked, rather than the total number of words stored. In this small example, the trie is less efficient than the BST since the BST requires at most 3 comparisons, while the trie requires up to 7. (However, note that each comparison in the BST is more complex, since you are comparing complete words instead of just letters.) As the size of the word list increases, the trie becomes more and more attractive.

3. **(28/25 pts)** Write a **TrieSpellChecker** class that supports the same functionality as your **BSTSpellChecker** from part (1). Specifically, it should include the following methods:

    a. *(11/10 pts)* **void add(String s)**
    Adds the specified word to the trie.

    b. *(11/10 pts)* **boolean contains(String s)**
    Returns whether the specified word is in the trie.

    c. *(6/5 pts)* A method that reads a list of words from a file and adds them all into the trie. In a trie, the order in which the words are added should not matter.

4. **(6/5 pts)** Within the same client program as part (2), create a new instance of **TrieSpellChecker** and experiment with its performance vs. the BST spell checkers.

Enhancing the Trie

Tries are also nice because they are well suited for supporting these features:

Autocomplete. You've certainly seen this in action on many websites and perhaps some IDEs. Suppose you want to generate a list of suggested words based on just the first few letters. You can simply search the trie for the first few letters, then look for all of the valid words that can be formed from that point onward. For example, starting with "abat" in the trie above would bring you to the 't' node on the left side. From there, the only two words that can be formed are "abate" and "abating."

5. **(16/15 pts)** Within the **TrieSpellChecker** class, write the method **autocomplete(String s)**. This should return a **Set<String>** of all the words in the trie that begin with the argument **s** (including **s** itself, if it happens to be in the trie).

Suggesting corrections for misspelled words. One way of doing this is to look for all words with an *edit distance* of 1 from the misspelled word. An edit distance of 1 means that you can transform the misspelled word into a correct word using a single insertion, replacement, or deletion of a character. For example, possible corrections of the word "ats" would include the following (where the ? indicates any letter):

- Insertion:

    ?ats
    a?ts
    at?s
    ats?

- Replacement:

    ?ts
    a?s
    at?

- Deletion:

    ts
    as
    at

Of course, not all of these suggestions should be made, since not all of them result in valid words. You can use the trie to help generate these suggestions by moving down the trie one step at a time, according to the letters in the misspelled word. At each step, consider the possibilities of adding a character at that position, replacing a character at that position, and removing a character at that position.

For example, with the word “ats” and the trie above, you would start from the root node and ask:

- “Can I add any character in front of ‘ats’ to produce a valid word?”
  The only three starting characters available in this trie are ‘a’, ‘b’, and ‘c’, so you would check “aats”, “bats”, and “cats”. Since “bats” and “cats” do exist in the trie, add them to the set of suggested corrections to return.
- “Can I replace the first character of ‘ats’ with anything else to produce a valid word?”
  Again, the only three starting characters available are ‘a’, ‘b’, and ‘c’, so you would check “ats”, “bts”, and “cts” and find that none of them exist.
- “Can I delete the first character of ‘ats’ to produce a valid word?”
  You would check “ts” and find that it does not exist.

Then you’d move down the trie according to the first character of “ats,” so you’re now at the top ‘a’ on the left side. You would repeat the procedure above, but this time focusing only on the part of the word after the first character (i.e., “ts”).

- “Can I add any character in front of ‘ts’ to produce a valid word?”
  From your current location (the top ‘a’ on the left side), the only next character available is ‘b’, so you’d check “bts” and find it doesn’t exist.
- “Can I replace the first character of ‘ts’ with anything else to produce a valid word?”
  Again, the only next character available is ‘b’, so you’d check “bs” and find that it doesn’t exist.
- “Can I delete the first character of ‘ts’ to produce a valid word?”
  You would check “s” and find that it doesn’t exist.

Keep repeating this until you’ve gone through all characters in “ats.” Each time you find a valid word, add it to the set to return.

6. **(23/21 pts)** Within the **TrieSpellChecker** class, write the method **closeMatches(String s)**. This should return a **Set<String>** of all the words in the trie that are within an edit distance of 1 from the argument **s**. Include **s** itself if it actually exists in the trie.

(10 pts) are set aside for coding style and documentation.

Need Help?

Please do not share code with anyone besides your teammate! If you find yourself stuck, feel free to contact me (Top) anytime. The Computer Science Learning Center in Dunn Hall 208 is also open, where you can get help from upper-division undergraduate students. Hours are posted at www.cs.memphis.edu/cslc.