Lab 8: Lexicon, Helicon, Lexical

CSCI 2101 – Fall 2018

Due: Tuesday, November 20, 11:59 pm
Collaboration Policy: Level 1
Group Policy: Pair-optional

Virtually all modern word processors contain a feature to check the spelling of words in documents. More advanced word processors also provide suggested corrections for misspelled words. In this lab, you will be undertaking the task of implementing such a spelling corrector, which will also have some other nifty features (such as performing pattern matching on text). To do so, you will build a recursive tree structure that is highly efficient at working with text prefixes, and augment it with various useful capabilities.

This lab will rigorously exercise your skills with recursion, trees, and layered abstraction in program design. This is probably the most challenging lab we have done all term – start early and work steadily!

1 Lexicon

A *lexicon* is defined as the vocabulary of a person, language, or branch of knowledge. More generally, it can be understood like a regular dictionary (i.e., a list of words), except that a lexicon contains only the words themselves (and not their definitions as well).

You will be writing a class that implements an interface for a `Lexicon` object. The complete `Lexicon` interface is provided to you, and its methods are sketched below:

```
public interface Lexicon extends Iterable<String> {
     boolean addWord(String word);
     boolean removeWord(String word);
     boolean containsWord(String word);
     boolean containsPrefix(String prefix);
     int addWordsFromFile(String filename);
     int numWords();
     Iterator<String> iterator();
     Set<String> suggestCorrections(String target, int maxDistance);
     Set<String> matchRegex(String pattern);
}
```

Most of these methods are self-explanatory and intuitive. For more details, refer to the `Lexicon.java` interface file provided in the starter files, which contains complete Javadoc for all methods in the interface. Don't worry too much about the final two methods for now (that is, `suggestCorrections` and `matchRegex`) – these will be implemented last, after the rest of the lexicon is working.

While the behavior of the lexicon is fairly straightforward, implementing it efficiently will require the use of a new type of tree structure that we have not seen before. This structure is described below.

2 Tries

There are several different data structures you could potentially use to implement a lexicon – a sorted array, a linked list, a binary search tree, and many others. Each of these options offers tradeoffs between the speed of looking up a word or prefix, the amount of memory required to store the data structure, the ease of writing and debugging the code, the performance of adding/removing, and so on. The structure that we will use is a special kind of tree called a *trie* (pronounced "try"), designed for just this purpose.

A trie is a letter-tree that efficiently stores strings. Each node in a trie represents a single letter. A path through the trie traces out a sequence of letters that represents a prefix or word in the lexicon. Note that the root of the trie is a special case, as it is 'blank' and does not represent a letter.

Instead of just two children as in a binary tree, each trie node potentially has **26 children** (one for each letter of the alphabet, ignoring case and non-letter characters). Whereas searching a binary search tree eliminates half the words with a left or right turn (we will discuss binary search trees in class shortly), a search in a trie follows the child reference for the next letter, which narrows the search to just words starting with that letter. For example, from the root node, any words that begin with "n" can be found by following the reference to the "n" child node. From there, following "o" leads to just those words that begin with "no" and so on, recursively. If two words have the same prefix, they share that initial part of their paths. This property can save significant space since there are typically many shared prefixes among words.

Clearly, leaf notes in a trie represent complete words (the path starting at the root node and ending at the leaf). However, note that since words may themselves be prefixes of other words (e.g., 'no' is a prefix of 'not'), internal nodes may or may not represent complete words. Thus, each node in the trie needs to know whether it represents a word or just a prefix.

A diagram of a small trie is shown in Figure 1. Nodes with a thick border are nodes representing words, while nodes with a thinner border are prefixes only. This trie contains the following seven words: `a, are, as, new, no, not,` and `zen`. Strings such as `ze` or `ar` are not valid words for this trie because the path for those strings ends at a prefix-only node. Any path not drawn is assumed to not exist, so strings such as `cat` or `next` are not valid because there is no such path in this trie.

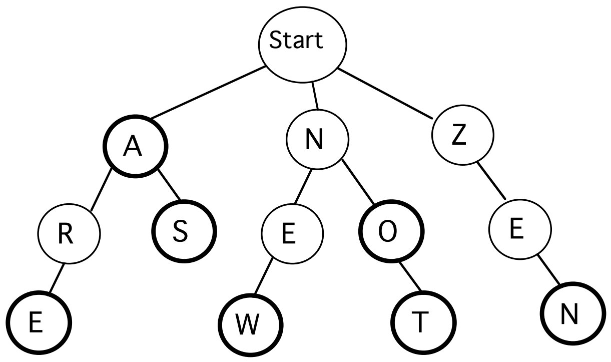

Figure 1: A sample trie storing 7 words.

A trie is an unusual data structure in that its performance can improve as it becomes more loaded. Instead of slowing down as its get full, it becomes faster to add words when they can share common prefix nodes with words already in the trie.

Like other trees, a trie is a recursive data structure, and all of the children of a given trie node are themselves smaller tries. You will be making good use of your recursion skills when operating on the trie!

2.1 Managing Node Children

For each node in the trie, you need to maintain references to the children nodes. For example, in Figure 1, the root node has three children, one each for the letters A, N, and Z. One possibility for storing the child references is a fixed-sized array of 26 references, where array[0] is the child for A, array[1] refers to B, ... and array[25] refers to Z. When there is no child for a given letter, (such as from Z to X) the array entry would be null. This arrangement makes it trivial to find the child for a given letter, as you simply access the correct element in the array by letter index.

However, for most nodes within a trie, very few of the 26 references are actually needed, so using a full 26-member array wastes a lot of space. A better alternative is a dynamic array (e.g., an `ArrayList`) or a linked list, both of which can grow and shrink as needed. While locating the correct child for a given letter will be more expensive with such a design, since there are at most 26 children per node (and usually far fewer), the added overhead is likely to be minimal.

2.2 Searching for Words and Prefixes

Searching the trie for words and prefixes is a matter of tracing out the path letter by letter. Let's consider a few examples on the sample trie shown previously in Figure 1. To determine if the string `new` is a word, start at the root node and examine its children to find one pointing to `n`. Once found, recurse on matching the remainder string `ew`. Find `e` among its children, follow its reference, and recurse again to match `w`. Once we arrive at the `w` node, there are no more letters remaining in the input, so this is the last node. Since this node is a word node, we know that this path represents a word contained in the lexicon.

Alternatively, consider searching for `ar`. The path exists and we can trace our way through all letters, but the last node is not a word node, indicating that this path is not a word. It is, however, a prefix of other words in the trie.

Finally, searching for `nap` follows `n` away from the root, but finds no child for `a` leading from there, so the path for this string does not exist in the trie. Thus, it is neither a word nor a prefix in this trie.

All paths through the trie eventually lead to a valid word node. Therefore, determining whether a string is a prefix of at least one word in the trie is simply a matter of verifying that the path for the prefix exists.

2.3 Adding Words

Adding a new word to the trie is a matter of tracing out its path starting from the root, as if searching. If any part of the path does not exist, the missing nodes must be added to the trie. In some situations, adding a new word will necessitate adding a new node for each letter. For example, adding the word `dot` to our sample trie will add three new nodes, one for each letter. On the other hand, adding the word `news` would only require adding an `s` child to the end of existing path for new. Finally, adding the word `do` after `dot` has been added doesn't require any new nodes at all, but does require changing a prefix-only node to a full word node.

Figure 2 depicts the sample trie from Figure 1 after these three words (`dot`, `news`, and `do`) have been added:

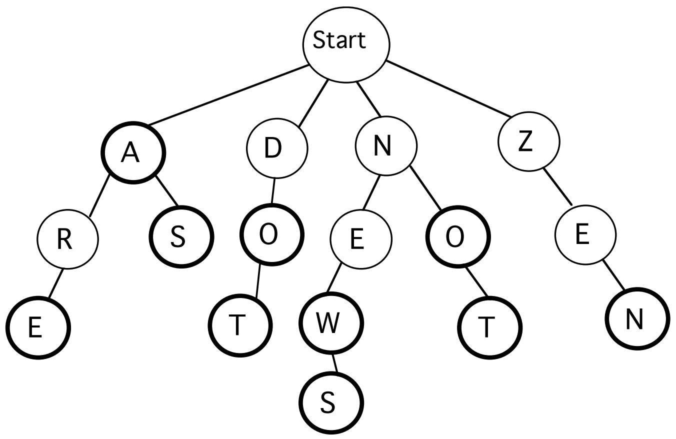

Figure 2: The sample trie after adding `dot`, `news`, and `do`.

2.4 Removing Words

The first step to removing a word is tracing out its path and changing the final node from a word node to a prefix-only node. While this will 'remove' the word from the trie, however, note that all the prefix nodes leading to the word are still present, and may now be dead ends.

To clean up the trie after removing the word, any nodes along the path that don't have other valid children must be deleted from the trie. For example, if you removed the words `zen` and `not` from the trie shown previously, you should end up with the trie shown in Figure 3:

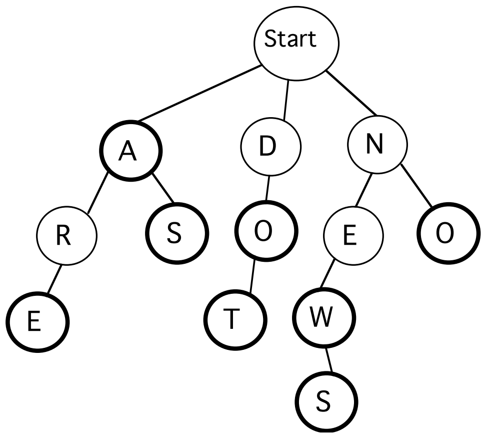

Figure 3: The sample trie after deleting `zen` and `not`.

Cleaning up nodes after deletion is an optional extension for extra credit. While just leaving old nodes in place after word deletion is okay, doing so does waste space and results in leaf nodes that don't represent actual words (as well as prefixes that aren't part of any complete words in the trie).

3 Advanced Lexicon Operations

This section details the final two `Lexicon` operations, `suggestCorrections` and `matchRegex`. These operations are challenging and should not be attempted until the rest of the `Lexicon` is working; as such, you may wish to revisit this section after implementing the basic operations.

3.1 Spelling Corrections

Consider the following strategy for suggesting corrections to a (potentially misspelled) word. Define the *distance* between two words of equal length as the number of character positions in which the words differ. For example, the words "place" and "peace" have distance 1, while the words "place" and "plank" have distance 2. Given a target word and a maximum distance (e.g., 3), we can suggest all words in the lexicon with distance to the target no greater than the maximum as possible corrections.

For example, consider the original sample trie containing the words `a, are, as, new, no, not,` and `zen`. If we were to call `suggestCorrections` with the following target string and maximum distance, here are the suggested corrections:

Target string	**Max distance**	**Suggested corrections**
ben	1	zen
nat	2	new, not

For a more rigorous test, consider a lexicon containing the full contents of the second edition of the Official Scrabble Player's Dictionary (aka OSPD2). Here are a few examples of calling `suggestCorrections` on this lexicon:

Target string	**Max distance**	**Suggested corrections**
crw	1	caw, cow, cry
zqwp	2	gawp, yawp
lexicon	2	lexicon, helicon, lexical

Finding appropriate spelling corrections in a trie requires a recursive traversal that gathers those "neighbors" that are close to the target path. In particular, you should *not* find suggestions by examining every word in the lexicon and seeing if it is close enough (you could, but doing so would be extremely slow). Instead, think about how you can generate candidate suggestions by traversing the path of the target string and taking small "detours" to the neighbors that are within the maximum distance.

3.2 Pattern Matching

One of the most powerful ways to locate patterns in text is using a *regular expression* (or *regex*), which is a string that specifies a search pattern (some of you may be having flashbacks to 1101 right now). You will implement a `matchRegex` method that allows matching simple regexes against all words in the lexicon.

The regular expressions your program will support may consist only of regular letters (which must match the corresponding letter in any matching word) and three different 'wildcard' characters, as specified below:

- The '_' wildcard character matches any one character.
- The '*' wildcard character matches any sequence of zero or more characters.
- The '?' wildcard character matches either zero or one characters.

For example, consider the original sample trie containing the words `a, are, as, new, no, not,` and `zen`. Here are the matches for some sample regular expressions:

Regular expression	**Matching words from lexicon**
a*	a, are, as
a_	as
a?	a, as
e	are, new, zen
_e_	new, zen
not	not
z*abc?*d	
*o?	no, not

Finding all words that match a regular expression will require applying your finest recursive skills. You should *not* find suggestions by examining each word in the lexicon and seeing if it is a match. Instead, think about how to generate matches by traversing the path of the pattern. For non-wildcard characters, the search proceeds just as for traversing ordinary words. For wildcard characters, "fan out" the search to include all possibilities for that wildcard (i.e., this will involve multiple recursive calls).

4 Class Overview

As in the last lab, some classes will be provided to you, while you will need to write others yourself.

You will be provided with the `Lexicon` interface (described in Section 1) and one regular class, the `TestLexicon` class. The tester class contains a `main` method that launches a text-based interface to interact with a `Lexicon` object. This class will be very useful when developing, debugging, and testing your trie. Using the tester program should be self-explanatory, and instructions are provided upon starting the program. You should not modify either the `Lexicon` interface or the `TestLexicon` class.

You are responsible for writing two additional classes: a `LexiconTrie` class, which is the actual implementation of the `Lexicon` interface, and a `LexiconNode` class, which represents a single node within the `LexiconTrie`. This approach is similar to that of our `SimpleLinkedList` class (more so than our `BinaryTree` class, in fact), in that there is a class representing the data structure itself (`LexiconTrie`) composed of multiple recursively-defined "node" objects represented by a second class (`LexiconNode`).

Note that the provided `TestLexicon` class constructs a `LexiconTrie` object to use (even though the `LexiconTrie` class does not initially exist). Thus, the `TestLexicon` class will not compile until you have a compiling skeleton of the `LexiconTrie` class in place.

In addition to the `Lexicon` interface and `TestLexicon` class, you will be provided with three sample data files that can be used in conjunction with the `addWordsFromFile` method to populate a trie. Two of the data files are small and better for initial testing, while the third is the complete Scrabble dictionary mentioned previously (OSPD2). This third data file is quite large and should probably not be used until you are ready to start stress-testing your program.

5 Implementation Plan

Below is a suggested plan of action for tackling the program.

- Start by implementing the `LexiconNode` class, which represents a single node in the lexicon trie. The skeleton of this class is likely to remind you of the `GameTree` class from Hexapawn. Make sure that you maintain all the state necessary for one each node of the trie by defining appropriate instance variables. The most important functionality that you will need to provide in this class is interacting with the child nodes (e.g., adding, getting, and removing children). One important note here is that since you will want to be able to traverse the trie in alphabetical order, it will be very helpful to keep the list of child references in alphabetical order (i.e., sorted by letter). Rather than resorting the entire list every time you add a child (which would be very slow), just insert each new child at the appropriate (sorted) position.

  As usual, your class should have any needed methods to work with the children – don't short-circuit your class by writing a getter that just returns the internal list of children.

  Finally, you may wish to have the `LexiconNode` implement the `Iterable` iterface and provide iteration capabilities over its children. Doing so will simplify the implementation of the `LexiconTrie` class by allowing you to use for-each style loops over `LexiconNode` objects. Implementing the `Iterable` interface should be very easy – you don't need to write a brand-new `Iterator` class, but can just recycle the iterator provided by the child list by calling its `iterator()` method.

- After completing `LexiconNode`, move on to `LexiconTrie`. To start, get a skeleton class in place so that you can run the `TestLexicon` program. The testing code makes calls to all of the public methods of the lexicon, but this doesn't mean that you should write all the code first and then attempt to debug it all at once (you shouldn't!). Instead, you can implement methods with placeholder "stubs" to start (i.e., trivial implementations that compile but just return dummy values). For example, if your lexicon doesn't yet remove words, implement a remove operation that just ignores its argument (or alternately, raises an error). Similarly, before you have implemented regular expression matching, just return an empty set from the method, and so on.

  In addition to the methods defined by the `Lexicon` interface, you'll need to add a constructor. The constructor should create a single `LexiconNode` representing the root. Assuming your `LexiconNode` objects are associated with characters (as they should be), you can have the root node be associated with a blank space ' ' (which is still a character just like any other).

- You should now be able to run `TestLexicon` to interact with your `Lexicon` methods. Now move onto implementing `containsWord` and `containsPrefix`. The technique used in both of these methods is basically the same (so you may want to consider writing a helper method that both of them use – always watch out for duplicate code blocks!). Note that you can implement these method either with or without recursion! Whichever way you prefer is fine.

- Now move onto `addWord` and `addWordsFromFile`. The latter should use a `Scanner` to parse the input file line-by-line, and should call `addWord` for each one. Remember to convert everything to lowercase before adding it to the trie. This is a good time to implement the `numWords` method as well.

  At this point, you should be able to run some non-trivial tests using `TestLexicon` to populate a trie from a data file, then test the `containsWord` and `containsPrefix` methods against strings in the data file (or not in the data file, as the case may be).

- This is probably a good time to implement the `iterator` method of the `LexiconTrie` class. As with the `LexiconNode` class, you do not need to write a brand-new iterator, but can just reuse an existing class' `iterator()` method. Things are somewhat trickier here, however, because you don't have an existing list that contains everything you want to iterate through. Instead, you'll need to populate such a list using a recursive traversal of the trie nodes. Keep in mind that assuming you followed the earlier advice, the `LexiconNodes` already maintain a list of their children in sorted order. That will help you iterate over the trie in alphabetical order. Remember that it is only words (not prefixes) that you want to operate on.

- The `removeWord` method may be implemented recursively or iteratively. If you choose to do it recursively, you may want to use a helper method. Remember that performing node cleanup (i.e., deleting unneeded nodes) is an optional extension; this method is simpler if you don't worry about cleaning up.

  If you do want to delete unneeded nodes, as a general observation, there should never be a leaf node that is a prefix node only. If a node has no children and does not represent a valid word, then this node is not part of any path to a valid word in the trie, and such nodes should be deleted when removing a word. In some cases, removing a word from the trie may not require removing any nodes.

  Lastly, note that when removing a word from the trie, the only nodes that may require removal are nodes on the path to the word that was removed. It would be extremely inefficient to check additional nodes that are not on the path.

- At this point, the entire `Lexicon` should be working except for the two advanced operations `suggestCorrections` and `matchRegex`. Put on your recursion hat and try to tackle them (in either order – they aren't dependent on each other). You will probably want to use helper methods for both.

- Once everything is working, try running tests on the Scrabble dictionary (especially using the advanced operations) – you can do some interesting things with the advanced operations on a large lexicon!

6 Implementation Advice

Here are specific tips about various parts of the program.

6.1 Debugging

Test early and often! Make extensive use of the provided tester program to exercise each method as you write it. Writing a bunch of interconnected classes and methods and trying to test all at once is a recipe for frustration. Also remember to use simple print-based debugging (i.e., printing key variables) when something isn't working but you're not quite sure where the problem is located.

6.2 Recursion Tips

Many of the methods of the `LexiconTrie` class may be recursive. However, the only methods that **must** be recursive are `matchRegex` and `suggestCorrections`. Other methods can be done recursively (and it may make sense to do so), but you will not be penalized if you choose to implement them iteratively. Use whichever technique you find most straightforward.

As a general note when writing your recursive methods, use extra parameters when you need to pass extra data through the recursive calls. For example, if you are writing a recursive method that is building up a list of values, you can pass the list as a parameter down the stack of recursive calls. As long as you're passing the same list through each call and not creating a brand-new list each time, any modifications to the list that the recursive calls make will "stick" once recursion is over (since they'll all be using the same actual list object, even though there are multiple distinct recursive calls).

You will probably have an easier time debugging your recursive methods if you stick to using parameters to pass data between recursive calls and avoid using instance variables (e.g., you could store a list in an instance variable and have your recursive calls modify that instead, but you're more likely to accidentally run into problems that way).

Finally, note that while the `LexiconNode` class is recursive in the sense that a trie node contains references to other trie nodes (i.e., its children), none of the methods of the `LexiconNode` class should be recursive.

6.3 Tree Traversals

Most of the recursion in this lab consists of recursively traversing the trie nodes, which is basically just a fancy way of saying getting to (or 'visiting') each node via a recursive call to that node. Following this approach, here's pseudocode for how one could recursively "visit" each node in the trie (or in any tree):

```
function visit(node):
    print("I'm here!")
    for each child of node:
        visit(child)
```

Of course, the above pseudocode doesn't do anything useful, whereas your recursive traversals are actually trying to accomplish something (e.g., collecting all the words stored in the trie for the `iterator` method of `LexiconTrie`). Thus, what you actually do when "visiting" each node will be more than just printing some dummy message as in the above example.

6.4 Sets

The two "advanced" lexicon operations are defined to return `Set` objects. A set is an abstract data type (and also a Java interface, like `List` or `Map`) that represents an unordered collection of elements in which duplicates are not allowed. The fundamental operations of a set are adding/removing an element and checking whether some element exists within the set. When working with a set, we often just want to iterate over its elements (which might be returned in some arbitrary order).

The implementation of the `Set` interface that you should use is the `HashSet` class. Consult the Javadoc for constructing and working with a `HashSet` object.

Note that one place we have seen `Sets` before (even if we didn't realize it at the time) is when we were using `Maps` – the collection of all keys for a given map (e.g., as returned by the `keySet` method) is a `Set`. This makes sense when we note that a map cannot have duplicate keys, and the keys exist in no particular order. It is not a coincidence that the standard `Map` implementation (i.e., `HashMap`) is named so similarly to the standard `Set` implementation (`HashSet`) – they use the same underlying technique (hashing), which we will learn about towards the end of the semester.

7 Evaluation

As usual, your completed program will be graded on correctness, design, and style. Make sure that your program is documented appropriately and is in compliance with the coding and style guide. Lastly, make sure that your name (and the name of your partner, if applicable) is included in all of your Java files.

8 Submitting Your Program

Submit your program on Blackboard in the usual way. Remember to create a zip file named with your username(s) and lab number, e.g., `sbowdoin-jbowdoin-lab8.zip`, and upload that file. Also remember to submit your group reports to me by email if working with a partner.