CSCI 136: Fall 2017
Handout L8
29 October

Lab 8

Due 5 November

Super Lexicon!

1 Prelab

Before lab, read through this entire handout carefully. Construct a design document for the `LexiconNode` class as described in Section 2.3. We will briefly go over this design together at the beginning of lab on Wednesday, and we will collect your design documents. Please bring a physical copy or a PDF document.

2 Lab Program

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. For your next lab, you will be taking on the fun and challenging task of implementing such a spelling corrector. You are to implement a highly-efficient Lexicon class and go on to augment it with additional functionality to find spelling corrections for misspelled words.

The assignment has several purposes:

- To gain further skill with recursion in managing a recursive tree structure and exploring it using sophisticated algorithms;
- To explore the notion of a layered abstraction where one class (Lexicon) makes use of other classes (*e.g.*, Iterator, Vector, Set) as part of its internal representation.

2.1 The Lexicon Interface

Your first task is to implement a basic `Lexicon` class. You may recall this class from the recursion lab (Lab 3), where an optional extension used `Lexicon.class` to provide word completions for T-9 word cell-phones (if you want to see such a cell phone "in the wild", Bill J has one). Now, you'll get a chance to build a lexicon yourself to see the kind of specialized and highly-efficient data structures that are used in its implementation. This version has similar functionality to the Lexicon from the Lab 3 extension, with additional operations to remove words, iterate over words, and read words from a text file. The following methods are in the `Lexicon` interface. You should implement this interface in a file called `LexiconTrie.java`.

```
public interface Lexicon {
     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);
}
```

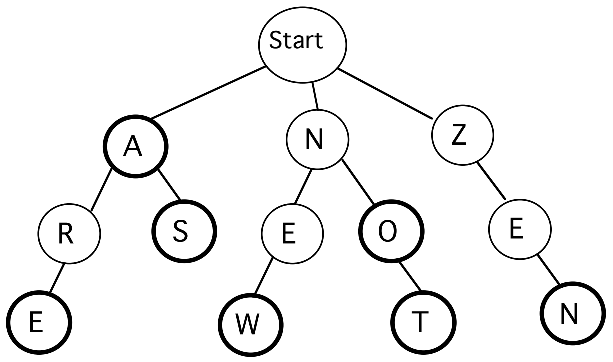
Most of the method behaviors can be inferred from their names and return types. For more information about the usage and purpose of these methods, refer to the comments in the starter code.

2.2 Implementing the Lexicon as a Trie

There are several different data structures you could use to implement a lexicon—a sorted array, a linked list, a binary search tree, and many others. Each of these offers tradeoffs between the speed of word and prefix lookup, amount of memory required to store the data structure, the ease of writing and debugging the code, performance of add/remove, and so on. The implementation 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. A node in a trie represents a letter. A path through the trie traces out a sequence of letters that represent a prefix or word in the lexicon.

Instead of just two children as in a binary tree, each trie node has potentially **26 child pointers** (one for each letter of the alphabet). Whereas searching a binary search tree eliminates half the words with a left or right turn (we will eventually discuss binary search trees in class), a search in a trie follows the child pointer for the next letter, which narrows the search to just words starting with that letter. For example, from the root, any words that begin with "n" can be found by following the pointer 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 saves space since there are typically many shared prefixes among words. Each node has a boolean `isWord` flag which indicates that the path taken from the root to this node represents a complete word. Here's a conceptual picture of a small trie:



The thick border around a node indicates that its `isWord` flag is true. This trie contains the 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 node where `isWord` is false. Any path not drawn is assumed to not exist, so strings such as `cat` or `astronaut` are not valid because there is no such path in this trie.

Like other trees, a trie is a recursive data structure. 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!

2.3 Managing Node Children

For each node in the trie, you need a list of pointers to children nodes. In the sample trie drawn above, the root node has three children, one each for the letters A, N, and Z. One possibility for storing the children pointers is a statically-sized 26-member array of pointers to nodes, 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—you simply access the correct element in the array by letter index.

However, for most nodes within the trie, very few of the 26 pointers are needed, so using a largely `null` 26-member array is much too expensive (i.e., it wastes a lot of space). Better alternatives would be a dynamically-sized array which can grow and shrink as needed (*i.e.*, a Vector), or a linked list of children pointers. We leave the final choice of a space-efficient design up to you, but **you should justify the choice you make (briefly) in your program comments**. Two things you may want to consider:

1. there are at most 26 children, so even a O(N) `List` operation to find a particular child is O(1), and
2. trie operations such as iteration require traversing the words in alphabetical order, so keeping the list of children pointers sorted by letter will be advantageous.

Begin implementing your trie by constructing a `LexiconNode` **class.** `LexiconNodes` should be Comparable, so make sure to implement the `Comparable` interface. After implementing `LexiconNode`, you should create a constructor in `LexiconTrie` that creates an empty `LexiconTrie` to represent an empty word list. **Be sure to incrementally compile and test your code.**

2.4 Searching for Words and Prefixes

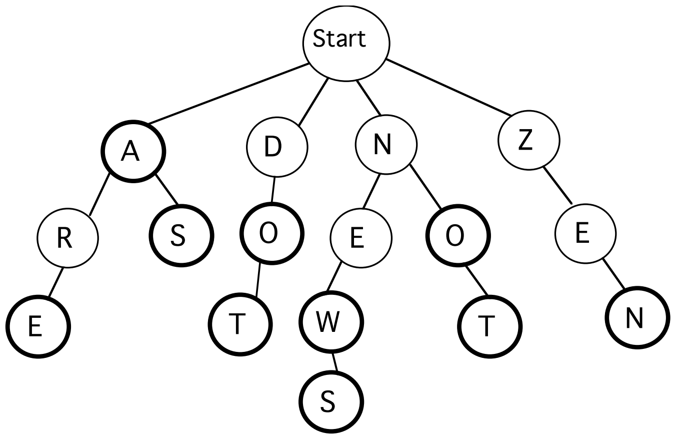
Searching the trie for words and prefixes using `containsWord` and `containsPrefix` is a matter of tracing out the path letter by letter. Let's consider a few examples on the sample trie shown previously. To determine if the string *new* is a word, start at the root node and examine its children to find one pointing to *n*. Once found, recurse on matching the remainder string *ew*. Find *e* among its children, follow its pointer, 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. The `isWord` field of this node is `true`, indicating that the path to this node is a word contained in the lexicon.

Alternatively, search for *ar*. The path exists and we can trace our way through all letters, but the `isWord` field on the last node is `false`, which indicates that this path is not a word. (It is, however, a prefix of other words in the trie). Searching for *nap* follows *n* away from the root, but finds no *a* child leading from there, so the path for this string does not exist in the trie and it is neither a word nor a prefix in this trie (`containsWord` and `containsPrefix` both return `false`).

All paths through the trie eventually lead to a valid node (a `LexiconNode` where `isWord` has value `true`). 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.

2.5 Adding Words

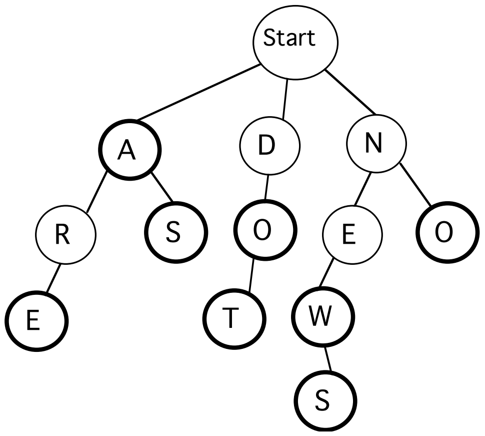
Adding a new word into the trie with `addWord` is a matter of tracing out its path starting from the root, as if searching. If any part of the path does not exist, the missing nodes must be added to the trie. Lastly, the `isWord` flag is turned on for the final node. In some situations, adding a new 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. Adding the word *do* after *dot* has been added doesn't require any new nodes at all; it just sets the `isWord` flag on an existing node to true. Here is the sample trie after those three words have been added:



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.

2.6 Removing Words

The first step to removing a word with `removeWord` is tracing out its path and turning off the `isWord` flag on the final node. However, your work is not yet done because you need to remove any part of the word that is now a dead end. (**This last part is left as an optional extra credit extension, described in the next paragraph.** You only need to update the `isWord` flag for full credit.) All paths in the trie must eventually lead to a word. If the word being removed was the only valid word along this path, the nodes along that path must be deleted from the trie along with the word. For example, if you removed the words `zen` and `not` from the trie shown previously, you should have the result below.



Optional extension. Deleting unneeded nodes is pretty tricky because of the recursive nature of the trie. Think about how we removed the last element of a `SinglyLinkedList` (Chapter 9.4 in *Bailey*). We had to maintain a pointer to the second to last element to update the pointers appropriately. The same is true in our trie.

As a general observation, there should never be a leaf node whose `isWord` field is false. If a node has no children and does not represent a valid word (*i.e.*, `isWord` is false), 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. For example, if we were to remove the word `new` from the above trie, it turns off `isWord` but all nodes along that path are still in use for other words.

Important note: 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.

2.7 Other Trie Operations

There are a few remaining odds and ends to the trie implementation.

1. You need to keep track of the total number of words stored in the trie.

2. You should add support for reading words from a file using a `Scanner`. You may find the **Scanner handout** on the course webpage helpful.

3. Creating an iterator to traverse the trie involves a **recursive** exploration of all paths through the trie to find all of the contained words. Remember that it is only words (not prefixes) that you want to operate on and that the iterator needs to access the words in alphabetical order. You may find the approach that we used in the **ReverseIterator.java** example from class helpful (e.g., creating a data structure that contains the elements in the desired order and returning an iterator for that data strurcture).

3 Optional Extensions

Once you have a working lexicon and iterator, you're ready to implement snazzy spelling correction features. There are two additional Lexicon member functions, one for suggesting simple corrections, and a second for regular expressions matching:

```
Set<String> suggestCorrections(String target, int maxDistance);
Set<String> matchRegex(String pattern);
```

Sets are basically just fancy Vectors that do not allow duplicates. Check out the javadocs on **Sets** and **SetVectors** in structure5 for more information.

3.1 Suggesting Corrections

First consider the member function `suggestCorrections`. Given a (potentially misspelled) target string and a maximum distance, this function gathers the set of words from the lexicon that have a distance to the target string less than or equal to the given maxDistance. We define the distance between two equal-length strings to be the total number of character positions in which the strings differ. For example, "place" and "peace" have distance 1, "place" and "plank" have distance 2. The returned set contains all words in the lexicon that are the same length as the target string and are within the maximum distance.

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, we also provide the word file ospd2.txt, which lists all of the words in the second edition of the Official Scrabble Player's Dictionary. Here are a few examples of `suggestCorrections` run on a lexicon containing all the words in ospd2.txt:

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

Finding appropriate spelling corrections will require a recursive traversal through the trie gathering those "neighbors" that are close to the target path. You should not find suggestions by examining each word in the lexicon and seeing if it is close enough. Instead, think about how you can generate candidate suggestions by traversing the path of the target string taking small "detours" to the neighbors that are within the maximum distance.

3.2 Matching Regular Expressions

The second optional extension is to use recursion to match regular expressions. The `matchRegex` method takes a regular expression as input and gathers the set of lexicon words that match that regular expression.

If you have not encountered them before, a regular expression describes a string-matching pattern. Ordinary alphabetic letters within the pattern indicate where a candidate word must exactly match. The pattern may also contain "wildcard" characters, which specify how and where the candidate word is allowed to vary. For now, we will consider a subset of wildcard characters that have the following meanings:

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

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 expression:

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

Finding the 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, it proceeds just as for traversing ordinary words. On wildcard characters, "fan out" the search to include all possibilities for that wildcard.

4 Suggestions

- Lexicon operations are case-insensitive. Searching for words, suggesting corrections, matching regular expressions, and other operations should have the same behavior for both for upper and lowercase inputs. Be sure to take that into consideration when designing your data structure and algorithms.
- ***Build and test incrementally***. Develop your trie one function at a time and continually test as you go. We have provided a handy client program that exercises the lexicon and allows you to drive the testing interactively from the console. It is supplied in source code form (`Main.java`) and you are encouraged to modify and extend it as needed for your purposes.
- Use stubs as placeholders. The testing code we provide makes calls to all of the public member functions on the Lexicon. In order for the program to compile, you must have implementations for all the functions. However, this doesn't suggest that you need to write all the code first and then attempt to debug it all at once. You can implement methods with placeholder "stubs" to start. If your lexicon doesn't yet remove words, implement the remove operation to ignore its argument or raise an error. Before you have implemented regular expression match, just return an empty set from the function and so on.
- Test on smaller data first. There are small.txt and small2.txt data files with just a few words that are especially helpful for testing in the early stages. You can also create word data files of your own that test specific trie configurations. The ospd2.txt word file is very large. It will be useful for stress-testing once you have the basics in place, but it can be overwhelming to try to debug using that version.

5 Thought Questions

Suppose we use an OrderedVector instead of a trie for storing our Lexicon. Discuss how the process of searching for suggested spelling corrections would differ from our trie-based implementation. Which is more efficient? Why?

6 Completing the Lab

We provide basic starter code for this assignment. To obtain it, execute the following command to copy the lexicon lab folder to your own directory:

```
cp -r /opt/mac-cs-local/share/cs136/labs/lexicon .
```

The lexicon directory contains the following files:

- `Lexicon.java`: The interface that you need to implement
- `Main.java`: Sample code that you can use to test your LexiconTrie
- `LexiconTrie.java`: skeleton for a LexiconTrie implementation
- `LexiconNode.java`: skeleton for the class that represents Trie nodes
- `small.txt`, `small2.txt`, `ospd2.txt`: Data files containing sets of words for you to test with

6.1 Deliverables

Create lab directory in the normal way (`<your_unix>-lab8/`) and submit it to the drop-off folder before the due date. Be sure to include:

- Your well-documented source code for all Java files that you write/modify.
- Your answer to the thought question in a file named `README.txt`.

As in all labs, you will be graded on design, documentation, style, and correctness. Be sure to document your program with appropriate comments, including a general description at the top of the file, a description of each method with pre- and post-conditions where appropriate. Also use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.

Have fun!