Assignment 5: Recursively Jumbled Tries

Due Date: 10:10 am, Tuesday, 14 April, 1998 in lecture

Note: For this assignment, you may work with one partner. Please hand in only one copy of the assignment. It should contain:

- Both of your names and CUIDs.
- The name of your recitation instructor, the section number, and the meeting time.
- The assignment title, and the date.

Introduction

One of the cultural highlights to living in Ithaca is that we are fortunate enough to have a truly world
class newspaper that carries the daily Jumble puzzle. In this puzzle, Ithacans are challenged daily to
unscramble four and five letter words, and to complete the witting remark that goes along with the cartoon.

In this assignment, you will build a program that might ruin Ithaca for good: you will build a data
structure called a trie (pronounced TRY) that stores a dictionary of words efficiently, and we will explore
how simple recursion can help us try all possible combinations of letters.

Part A: If at first you don’t succeed ...

This part does not require you to hand in anything. However, you will need to use the ideas in part C,
and it will give you some practice in dealing with recursion, so you should try it out.

Consider the following Java class (available from the assignment web page):

class StringPerm {
    public static void printPerms (String charList) {
        System.out.println("Printing permutations of "+charList);
        rPrintPerms(charList, "");
    }
    
    private static void rPrintPerms (String charList, String soFar) {
        final int N = charList.length();
        if (N==0) {
            System.out.println(soFar);
        } else {
            for (int i=0; i<N; i++) {
                String charListMinusIthChar = charList.substring(0,i) + charList.substring(i+1,N);
                rPrintPerms (charListMinusIthChar, soFar+charList.charAt(i));
            }
        }
    }
    
    public static void main (String[] args) {

1The copyright to Jumble is owned by some big, powerful media company, who might get upset if we left off this disclaimer or
characterized Jumble as anything other than truly wonderful and spiritually uplifting.
printPerms("a");
printPerms("be");
printPerms("tin");
printPerms("arts");
// printPerms("perks"); // try me last.

The function printPerms will print all possible permutations of a set of letters given as an argument. It does so by recursively calling a helper function rPrintPerms using an extra parameter keeps track of the current state of the computation. At each step, a for loop adds each possible next character to soFar at the same time removing it from the original list charList.

As an exercise, run the main function and see what is printed. Then try tracing through the execution of each invocation of printPerms by hand with pen and paper.

You’ll quickly notice that the computational complexity of printPerms is factorial in the length of the argument. That is, if string charList has length N, then the running time is \( O(N!) \). Usually, we do not solve a problem by just enumerating all of the possibilities, as it takes far too long for non-trivial data sets. However, Jumble puzzles typically scramble words of only a few letters, and \( 6! = 720 \) which is a pretty small number, so we will use the simple and naive approach for this assignment ...

Part B: ... trie, trie again

A trie is a kind of tree that stores sequences of data, such as character strings, in a way that makes searching efficient. The word trie comes from reTRIEval, but is pronounced TRY to distinguish it from other kinds of trees.

A trie can be used to implement a spelling dictionary, or lexicon. Each node stores one letter, plus an array of 26 references, one for each possible next letter. You can consider that tracing a path through a trie from root to leaf spells a word.

Additionally, each node stores a boolean value that indicates if the node is the last letter in a word (that is, if the node is a leaf or a prefix of a longer word).

The root node is special; it stores a space as its character value (think: why?).

The diagram below shows a trie into which only the following words have been added: for fork top tops topsy toss. Note that although "to" is a perfectly valid English word, we have not added it to the lexicon yet, and so the boolean field in the ‘o’ node is currently set to false. If you were now to add "to" to the lexicon, this field would be set to true.

Also, note that all of the nodes currently in the trie are shown in the diagram. Assume all references are null unless there is an arrow shown.
In this diagram, as in your program, nodes should be created only as they are needed. Initially, the lexicon contains only the root. As new words are added, new nodes are created. In particular, the array of 26 nodes for the children should initially be null for each node.

Below is an outline of the class \texttt{TrieNode}.

\begin{verbatim}
class TrieNode {
    public static final int NumLetters = 26;
    public static final String Alphabet = "abcdefghijklmnopqrstuvwxyz";

    char letter;
    boolean isWord = false;
    TrieNode[] children = new TrieNode[NumLetters];

    public TrieNode (char letter) {
        this.letter = letter;
    }

    // Note: this is correct but horribly inefficient if we are going to use
    // it a lot (which we are). For CS211, it is a reasonable solution.
    // In the real world, we would use a trick of some kind, such as:
    // return c - 'a';
    private int charIndex (char c) {
        return Alphabet.indexOf(c);
    }

    // You may decide to add more methods here.
}
\end{verbatim}
Your job is to write a Java class called `Lexicon` that implements a trie to store a set of strings. It should use `TrieNode` much in the same way that a `BinarySearchTree` class might use the `BSTnode` class shown in lecture.

Below are the methods of `Lexicon` that you must implement. Likely you will also need to add some functions to `TrieNode` too. We will leave these up to you. With some ADTs, it isn’t always clear which methods belong in the node class (e.g., `BSTnode`, `TrieNode`) and which in the ADT class (e.g., `BST`, `Lexicon`). Don’t worry too much about this issue; just do something reasonable.

- **public void `insertWord` (String s)** — insert the indicated word into the lexicon. Create nodes as needed and be sure to set the boolean on the last node to true. If a word has already been entered, don’t worry. You may assume all words consist only of lower case letters.

- **public void `addFileContentsToLexicon` (String inputFileName)** — add each word in the indicated file to the lexicon. Use `TokenReader` and its `readString` method to do this. Again, you may assume all words consist only of lower case letters. Have a look back at assignment #1 to see how you can go from a file name to something you can pull tokens out of. This method needn’t be recursive, but it will contains repeated calls to `insertWord`.

- **public boolean `hasWord` (String s)** — return true iff the indicated word is contained in the lexicon.

- **public void `print` ()** — print all of the words currently in the lexicon.

  *Hint:* Start at the root and recursively examine all of the children and their subtrees. As you are recursing through the trie, use an extra string parameter to keep track of the string defined by the path from the root to the current node. Start this parameter as the empty string and add the current character with each recursive call.

Be sure to use recursion to implement `insertWord`, `hasWord`, and `print`. Likely, these methods will not be recursive themselves, but will call a recursive “helper” method using the root as an extra argument.

Note also that part C will require another function or two to be implemented.

**Part C: Solving Jumble**

(This part is easy, one you have examined part A and solved part B!)

Adapt the functions given in part A to help solve the Jumble puzzle. That is, implement an additional public function as part of `Lexicon` that takes one string argument and prints all permutations of those letters that are found in the lexicon (you will use a private helper function to do this). Show some interesting and thorough testing for your function.

So for example, if your lexicon contains only the following words: post stop pots spot pot top opt stoop, then your function called with the argument "ptos" will print the words post stop pots spot but not ptos nor tsop² nor any other combination of four letters that is not contained in the lexicon.

---
²See also What to hand in.
A couple of notes:

- If you have duplication of any letters in your argument, some (valid) words may be printed more than once. This is fine. Don’t worry about it.

- It doesn’t matter what order the words are printed in. However, you might want to think about how you could generate an alphabetized list. (Good prelim question.)

What to hand in

- Complete code listing for all changed and/or new classes, including test suites. Check the web page for the test suites we expect you to use.

- Output from test runs of your own design to show that your implementations work.

Questions to ponder

Below are some interesting points to ponder about tries. You don’t have to answer these questions for the assignment, but you should think about them (again, good prelim fodder):

- Unlike binary search trees, the structure of a trie is independent of the order in which items are added. Why?

- A sorted linked list is often used instead of an array to store the node’s children. Why?

- After some thought you might realize that the letter stored in the trie node is actually redundant if you use an array for the children! You could derive the information merely by the position in the array of the parent. That is, you know the node following the ‘r’ has value ‘k’ since the parent points to it with child number 10, and ‘k’ is the 10th letter of the alphabet if you start at ‘a’ being zero.

- What is the complexity of lookup in a trie if you use an array? ... if you use a linked list? (This is subtle and requires some thought.)

- How much space do you waste by using an array?

And here’s the bonus question (for fun only, don’t think too hard about it):

- In the 1970s, a music group had a hit with a song called TSOP. What does this acronym stand for? The band was also known by an acronym. What was it and what does it stand for?