COMP 250: Introduction to Computer Science   Fall 2016

Assignment #3

Prepared by Prof. Michael Langer
Posted: Wed Oct 26
Due: Tues Nov 8, 23:59 PM

General Instructions

- The T.A.s handing this assignment are Stephanie Laflamme, Victor Chenal, Tzu-Yang (Ben) Yu, Pierre Thodoroff, and Rohit Verma. Their office hours and location and email contacts are posted on mycourses (see Announcements). Stephanie is the team leader, so if you have general issues about the assignment then please contact her or me.
- Same general instructions as Assignments 1, in particular, the late penalty.
- *We will deduct at least 20 points for any student who has to resubmit after the due date (i.e. late) because the wrong file or file format was submitted.* This policy will hold regardless of whether the student can provide proof that the assignment was indeed "done" on time. This includes submitting a .class file, starter code, a rar or 7z format.

Submission Instructions: **Submit a single zipped file A3.zip**, which contains the modified **Trie.java** file, to the myCourses assignment A3 folder. Include your name and student ID number in the comment at the top of the **Trie.java** file. If you have any issues that you wish the TAs (graders) to be aware of, then include them as a comment at the top of your file.

Question 1 Implementing Autocomplete with a Trie (70 points)

In this question, you will work on the problem of storing a set of *n* words (also called *keys*) in a tree data structure, and a method for efficiently finding all the words that contain a given **prefix**. By *word* or *key* here, we just mean a string formed from some set of ASCII characters. (If you are unfamiliar with ASCII, see http://www.asciitable.com/. ASCII encodes text using one byte or 8 bits per character.)

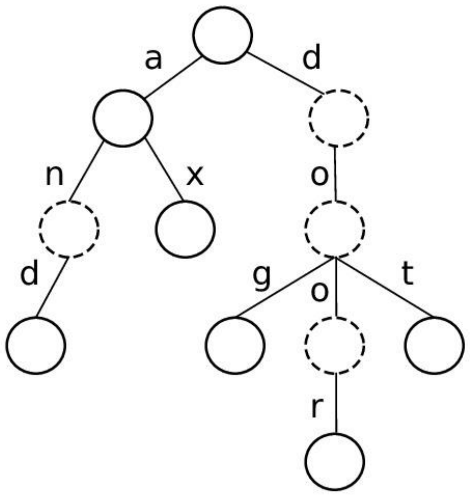
You should be familiar with the above problem through the autocomplete feature found on some cell phones, web forms, Eclipse. For instance, if you type in "*aar*", then autocomplete may suggest *aardvark, aardvarks, aardwolf, aardwolves, aargh*.

Trie Data Structure

A **trie** is a type of rooted tree used to efficiently index keys by storing their prefixes. (The term trie comes from the word "retrieval". It is sometimes pronounced "try" although many people pronounce it the same as "tree".) The main property of a trie is that *each edge corresponds to a character*. Thus the path from the root of the trie to any node in the trie defines a string. The string defined by each node is a prefix of all strings defined by the descendents of that node.

Below is an example trie. It contains the following keys: *a, and, ax, dog, door, dot.* You will note that the dashed nodes correspond to prefixes in the trie that are not in our list of keys (*an, d, do,* *doo*). The trie must also keep track of this distinction by storing for each node whether it corresponds to a key or not.

Note that if *C* is the set of all possible characters defined at the edges, then each node can have at most *k* = | *C* | children, where the | C | notation just means the number of elements in set C.



What You Need To Do

Read the provided code (***Trie.java, AutoComplete.java***), including the comments which explain what the various methods do. *This will take you some time, so go slow and read the code carefully.* Note that the ***Trie*** class has a private inner class ***TrieNode***.

For the ***Trie*** class, fill in the missing code for the following methods:

- *createChild(), toString()* for the inner class ***TrieNode***
- *getPrefixNode(), insert(), contains(), getAllPrefixMatches()*

We suggest you implement them in the order listed above. You may write helper methods, but if you do then add a comment explaining what you are doing so that the grader can easily follow.

You should use Java String methods *length()* and *charAt()*. But you may not use String searching methods such as *String.startsWith(), String.substring()*, etc. The point of the assignment is for you to organize and compare strings use a trie data structure.

Submit only the file ***Trie.java*** as a zip file. Use the *AutoComplete.java* file to test your code, but do not submit this tester file.

Question 2   Running time analysis of Tries (30 points)

Here you will analyze the complexity of two methods, namely *loadKeys()* and *contains()*.

Suppose you are given a set of *K* keys, where each key is of length at most *L* characters.   Let the keys be represented by a trie T_1 as implemented in Question 1. Let *C* be the maximum number of children of a node. Although C is a constant NUMCHILDREN in the code, we are asking you to treat it as a variable in these questions.

Answer the following questions. In each case, give the tightest bound.   For example, if the bound was O(*L*) and you wrote O(*KL*), then your answer would not be the tightest bound.   In addition, you must *provide a short written justification for your answer*.

In each case, your answer should be in terms of the variables *K, L, C*.

1) What is the O( ) bound of *loadKeys()* on T_1?
2) What is the O( ) bound of *contains()* on T_1?

In the ***TrieNode*** *inner class*, the children of a node is implemented using an array of size C. In many tree data structures, however, the set of children is implemented using a linked list. Suppose we were to modify the ***TrieNode*** inner class so that each node's children were implemented using a linked list. Let T_2 denote the modified trie data structure.

3) What is the O( ) bound of *loadKeys()* on T_2 ?
4) What is the O( ) bound of *contains()* on T_2 ?

We next compare the runtime of tries to the runtimes of binary search trees (BST). Consider a BST data structure that stores the same set of *K* keys. Each node in this BST would correspond to a key, such that the left (or right) child of each node is lexicographically smaller (or bigger) than its parent. Note there is no prefix representation with a BST.

Note that the running time of operations on a BST depends on how "balanced" the tree is. (Roughly speaking, a tree is well balanced if the number of descendents of each left child is approximately the same as the number of descendents of each right child.)

5) What is the O( ) bound of *loadKeys*() for a BST ?
6) What is the Ω ( ) bound of *loadKeys()* for a BST ?
7) What is the O( ) bound of *contains()* for a BST ?
8) What is the Ω ( ) bound of *contains()* for a BST ?

Assume that these methods perform the same role as they do for tries i.e. *loadKeys()* populates a BST, and *contains()* searches for a key in BST.

Hint: Comparing the lexicographic order of two strings of length L requires at least one and at most L comparisons.

What You Need To Do

Submit a text file A3Q2.txt with answers to the questions. Feel free to write Omega( ) instead of Ω ( ).   Include this text file in your A3.zip folder which you submit.