Programming Assignment 3

Trie

Posted Fri, Oct 21

Due Fri, Nov 4, 11:00 PM (WARNING!! NO GRACE PERIOD)

Extended deadline (with ONE time extension pass): Mon, Nov 7, 11:00 PM (NO GRACE PERIOD)

Worth 75 points (7.5% of your course grade)

In this assignment you will implement a tree structure called Trie (pronounced "Tree"!!).

You will work on this assignment individually. Read DCS Academic Integrity Policy for Programming Assignments - you are responsible for abiding by the policy. In particular, note that "**All Violations of the Academic Integrity Policy will be reported by the instructor to the appropriate Dean**".

- You get ONE extension pass for the semester, no questions asked. There will be a total of 5 assignments this semester, and you may use this one time pass for any assignment except the last one which will NOT permit an extension. **A separate Sakai assignment will be opened for extensions AFTER the deadline for the regular submission has passed.**

- **IMPORTANT - READ THE FOLLOWING CAREFULLY!!!**

Assignments emailed to the instructor or TAs will be ignored--they will NOT be accepted for grading.
We will only grade submissions in Sakai.

If your program does not compile, you will not get any credit.

Most compilation errors occur for two reasons:

1. You are programming outside Eclipse, and you delete the "package" statement at the top of the file. If you do this, you are changing the program structure, and it will not compile when we test it.
2. You make some last minute changes, and submit without compiling.

To avoid these issues, (a) START EARLY, and give yourself plenty of time to work through the assignment, and (b) Submit a version well before the deadline so there is at least something in Sakai for us to grade. And you can keep submitting later versions (up to 10) - we will accept the LATEST version.

Summary

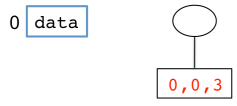
You will write an application to build a tree structure called Trie for a dictionary of English words, and use the Trie to generate completion lists for string searches.

Trie Structure

A Trie is a general tree, in that each node can have *any* number of children. It is used to store a dictionary (list) of words that can be searched on, in a manner that allows for efficient generation of completion lists.

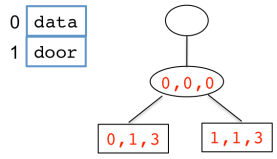
The word list is originally stored in an array, and the trie is built off of this array. Here are some word lists, the corresponding tries, followed by an explanation of the structure and its correspondence to the word list.

Trie 1



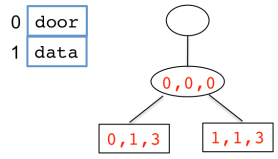
Root node is always empty. Child [0,0,3] of root stores "data" in a triplet 0 (for index of word in list), 0 (for position of first character, 'd') and 3 (for position of last character, 'a')

Trie 2



Child (0,0,0) of root stores common prefix "d" of its children "data" (left child) and "door" (right child), in triplet 0 (index of first word "data" in list), 0 (starting position of prefix "d"), and 0 (ending position of prefix "d"). Internal nodes represent prefixes, leaf nodes represent complete words. The left leaf node stores triplet 0 (first word in list), 1 (first index past the common prefix "d", and 3 (last index in word). The right leaf node is stored similarly.

Trie 3

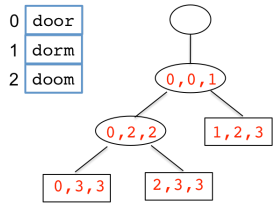


Like in trie 2, child of root stores common prefix "d", but this time left child is "door", and right child is "data", because "door" appears before "data" in the array.

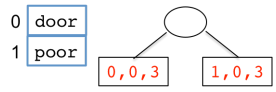
Trie 4

Trie 5

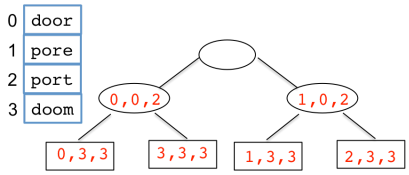
Trie 6



A node stores the longest common prefix among its children. Since "do" is the longest common prefix of all the words in the list, it is stored in the child of the root node as the triplet (0,0,1). The left branch points to a subtree that stores "door" and "doom" since they share a common prefix "doo", while the right branch terminates in the leaf node for "dorm" stored as the triplet 1 (index of word "dorm"), 2 (starting position of substring "rm" following prefix "do"), and 3 (ending position of substring "rm")

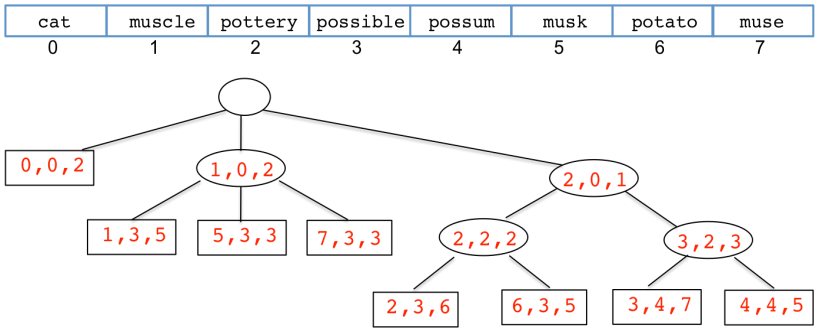


There is no common prefix in "door" and "poor", so the root has 2 children, one for each word. (Common suffixes are irrelevant)



There is no common prefix among all the words. But "door" and "doom" have a common prefix "doo", while "pore" and "port" have a common prefix "por".

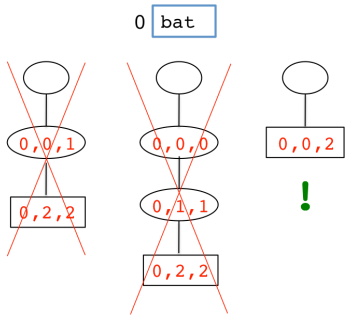
Trie 7



Special Notes

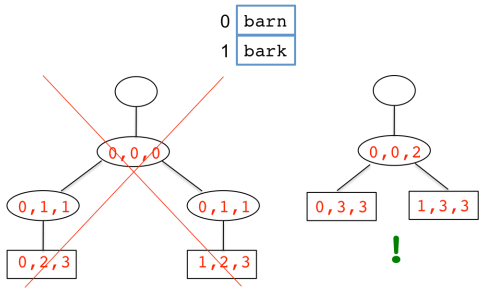
- No node, except for the root, can have a single child. In other words, every internal node has at least 2 children. Why? It's because an internal node is a *common prefix* of several words. Consider these trees, in each of which an internal node has a single child (incorrect), and the equivalent correct tree:

One-word trie
Incorrect/Correct



A single leaf node only

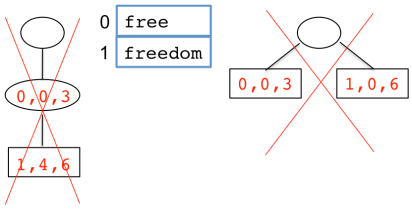
Two-word trie
Incorrect/Correct



The longest common prefix of the two words is "bar", so there is one internal node for this, with two branches for the respective trailing substrings

- A trie does NOT accept two words where one entire word is a prefix of the other, such as "free" and "freedom".

  The process to build the tree (described in the **Building a Trie** section below), will create a single child of the root for the longest common prefix "free", and this node will have a single child, a leaf node for the word "freedom". But this is an incorrect tree because it will (a) violate the constraint that no node apart from the root can have a single child, and (b) violate the requirement that every complete word be a leaf node (the complete word "free" is not a leaf node).



On the other hand, a tree with two leaf node children off the root node, one for the word "free" and the other for the word "freedom" will be incorrect because the longest common prefix MUST be a separate node. (This is the basis of completion choices when the user starts typing a word.)

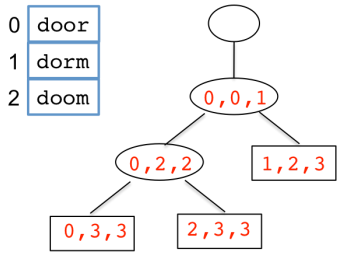
Data Structure

Since the nodes in a trie have varying numbers of children, the structure is built using linked lists in which each node has three fields:

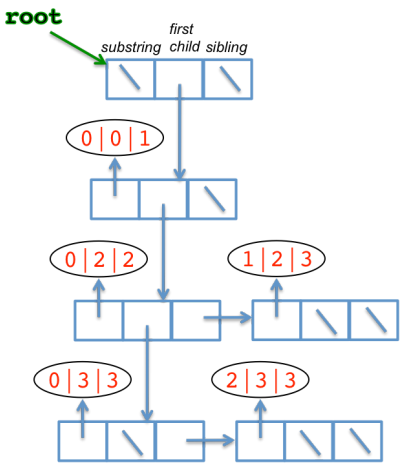
- **substring** (which is a triplet of indexes)
- **first child**, and
- **sibling**, which is a pointer to the next sibling.

Here's a trie and the corresponding data structure:

Trie



Data Structure



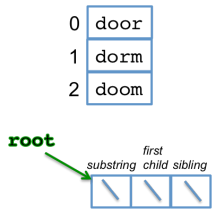
Building a Trie

A trie is built for a given list of words that is stored in array. The word list is input to the trie building algorithm. The trie starts out empty, adding one word at a time.

Example 1

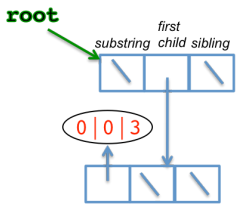
The following sequence shows the building of the above trie, one word at a time, with the complete data structure shown after each word is inserted.

Input and Initial Empty Tree



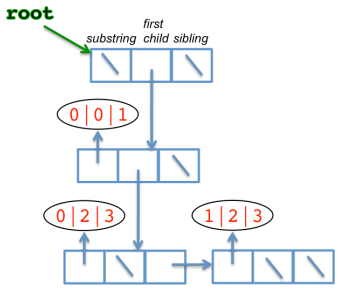
An empty trie has a single root node with nulls for all the fields.

After Inserting "door"



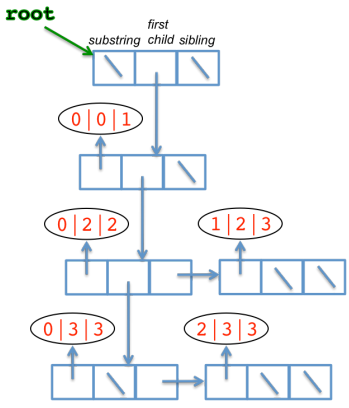
When "door" is inserted, a leaf node is created and made the first child of the root node. The substring triplet is (0,0,3), since "door" is at index 0 of the word list array, and the substring is the entire string, from the first position 0 to the last position 3.

After Inserting "dorm"



When "dorm" is inserted, its prefix "do" is found to match with prefix "do" in the existing word "door". So the third value in the triplet for the existing node is changed from 3 to 1, corresponding to the prefix "do". (The word index--first value in triplet--is left unchanged.) And two new nodes are made at the next level for the two trailing substrings, "or" of "door" and "rm" of "dorm" - The array indexes of these words are in ascending order, i.e. "door" must come before "dorm" in the node sequence.

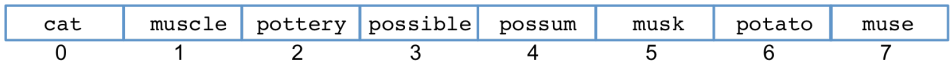
After Inserting "doom"

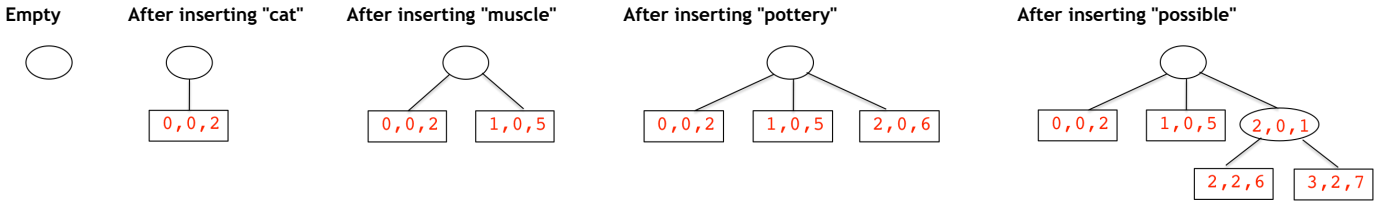


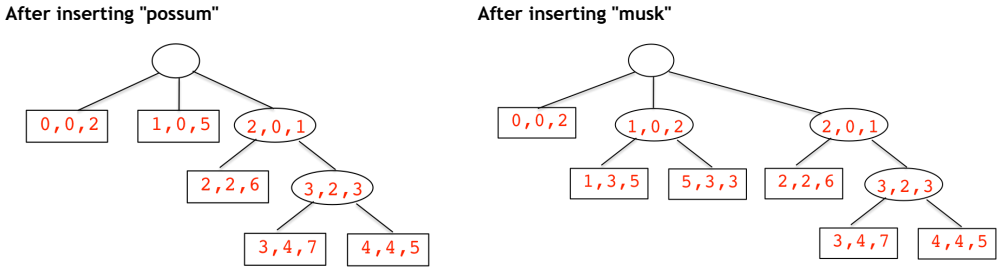
When "doom" is inserted, its prefix "do" is found to match with the entire substring stored at the child of the root. Descending further, the subsequent "o" is found to match with the prefix "o" of the substring "or" at the (0,2,3) node. This results in a modification of the (0,2,3) triplet to (0,2,2), and the creation of a new level for the trailing substrings "r" and "m" of "door" and "doom", respectively, in that order - "door" (word index 0 in array) must precede "doom" (word index 3).

Example 2

This shows the sequence of inserts in building Trie 7 shown earlier.







Implementation

Download the attached trie_project.zip file to your computer. DO NOT unzip it. Instead, follow the instructions on the Eclipse page under the section "Importing a Zipped Project into Eclipse" to get the entire project into your Eclipse workspace.

You will see a project called Trie Assignment with the following classes:

- structures.TrieNode
- structures.Trie
- apps.TrieApp

There are also a number of sample test files of words directly under the project folder (see the **Examples** section that follows.)

You will implement the following methods in the Trie class:

- **(50 pts)** insertWord: Inserts a word into the trie. Input word may be in any combination of upper and lower case letters (only letters, no other characters), but in the tree, all words are stored in lower case only.
- **(25 pts)** completionList: Gathers and returns the completion list of words for a given prefix. The order in which the words appear in the list is irrelevant.

Observe the following rules while working on Trie.java:

- You may NOT add any import statements to the file.
- You may NOT add any fields to the Trie class.
- You may NOT modify the headers of any of the given methods.
- You may NOT delete any methods.
- You MAY add helper methods if needed, as long as you make them private.

Also, do NOT change TrieNode.java in any way. When we test your program, we will use the original TrieNode.java as supplied with the assignment.

Testing

You can test your program using the supplied TrieApp driver. It first asks for words to be stored in the trie (either individually, or from a file), and when done, asks for prefixes for which to compute completion lists.

Some sample word files are given with the project, directly under the project folder. The first line of an input word file is the number of the words, and the subsequent lines are the words, one per line.

There's a convenient print method implemented in the Trie class that is used by TrieApp to output a tree for debugging purposes. (Note: Our testing will NOT look at this output - see the **Grading** section below.)

Submission

Submit your **Trie.java** file ONLY.

Grading

The insertWord method will be graded by inserting several words in sequence, and *after each insert, comparing the tree structure resulting from your implementation, with the correct tree structure produced by our implementation.* **We will NOT be looking at the printout of the tree,** the print method in the Trie class is for your convenience only.

The completionList method will be graded by inputting prefix strings to some of the trees created in insertWord. However, these *trees will be created by our correct implementation of* insertWord. In other words, to test your completionList implementation, we will NOT use your insertWord implementation at all. This is fully for your benefit, because if your insertWord implementation is incorrect, it will not adversely affect the credit you get for your completionList implementation.