CE204 ASSIGNMENT 2 2019

Set by: Mike Sanderson

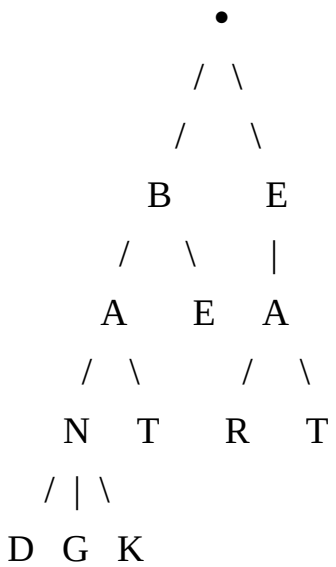
Credit: 10% of total module mark

Deadline: 11.59.59, Tuesday 19 March

Introduction

You should refer to sections 5 and 7 of the Undergraduate Students' Handbook for details of the University policy regarding late submission and plagiarism; the work handed in must be entirely your own.

A *trie* is a tree used for storing data items that are sequences, typically strings. Each path from the root to a leaf represents a data item. The diagram shows a trie containing the words BAND, BANG, BANK, BAT, BE, EAR and EAT.



Each node other than the root will contain an element of a sequence (in the example a character from a string) and a reference to children. The root will hold no data.

Part 1

Write a class called `Trie` to store words using the format in the diagram using a separate node class to represent the non-leaves. Each node object must contain a character and references to 0 or more children, each of which holds a different letter. A number of possible approaches can be used, e.g. a linked list of references to child nodes, a left-child right-sibling implementation as described at the end of part 3 of the lecture slides or a map where the keys are the characters in the children and the values are references to the children. (You are not required to use one of these options; you may if you wish devise a different alternative.) You may choose whether to store the children of a node in alphabetical order; doing so will make some of the required methods easier, whereas not doing so may make other methods easier.

The root node should be stored in the `Trie` object; it will not contain any character. If you are using a left-child right sibling approach it should contain a reference to the first child; otherwise an array or `ArrayList` of 26 possible children might be more appropriate.

Your class must be called `Trie` and must contain the following public methods:

`Trie()`:
A constructor that initialises the trie to be empty (i.e. to comprise a root with no children; if you are using an array of 26 children the array should be initialised to hold 26 null references)

`boolean addWord(String word)`
A method to attempt to add a word to the trie, which will return `true` if the word is added successfully, and `false` if the word cannot be added. The word cannot be added if it is already present; additionally the method should check that the string contains only letters, and output a message and return false if it contains any non-letters.

`boolean find(String word)`
a method to check if a word is present in the trie

`List<String> getWords(char c)`
A method that returns a list of all words in the trie that begin with the letter specified by the argument, sorted alphabetically. An empty list should be returned if there are no words beginning with the specified letter, or if the argument is not a letter, in which case a message should also be output. You may return an object of any class that implements the `List` interface.

The methods of the class should not perform any input or output other than the output of the non-letter messages. They must have the names and argument types specified here to allow my test program to compile with your class.

The letters in the trie should not be case-significant so you should convert all letters obtained from method arguments to either upper or lower case.

All data in the `Trie` class must be private. You may write additional private support methods if you wish and may also write methods in the node class. (The name of the node class does not matter.)

An additional method will be required for part 3; if you do not attempt part 3 you must add to your class a method `boolean delete(String word)`, that outputs a message saying that part 3 was not attempted and returns false.

It is strongly recommended that you write an extra class with a main method to test your class before proceeding to parts 2 and 3, but you should not submit this and there must be no main method in the `Trie` class.

Part 2

Since words are currently represented as paths from the root to a leaf, words that are prefixes of other words in the trie cannot be stored. In the example we may wish to store the word BAN, but its path is already present and does not end in a leaf.

Make modifications to your class to allow words that are prefixes of other words to be stored. There are a number of possible approaches that could be used. The following are suggestions but you may use a different approach if you wish

1. Store in each non-leaf node an extra boolean member to indicate whether it is the end of a word
2. If a non-leaf node denotes the end of a word add to it an extra child, containing a character such as `'*'`
3. Store characters in non-leaf nodes that are the end of a word using the opposite case to that used for other letters (remembering to convert everything back to a single case in the list returned by the `getWords` method)

Part 3

Add to the `Trie` class a public method `boolean delete(String word)`, which should attempt to find and delete the word specified in the argument. The return value should indicate whether it was successfully removed. As in the earlier methods, this method should not be case-significant and if the argument contains any characters that are not letters a message should be output before returning `False`.

If the word to be deleted is a prefix of another word in the trie you will need to undo whatever was done in part 2; otherwise any nodes containing letters from this word that are not part of other words should be removed.

Comments and Documentation

You should provide at the beginning of the `Trie` class comments stating how you have represented the list of children of a node, and (if you have attempted part 2) what technique you have used indicate that a non-leaf is the end of word.

Above each method in every class there should be a comment describing precisely what it does and returns and what its arguments are. Within the method bodies you should provide only brief comments describing what blocks of code do.

Marking Scheme

45% of the marks for the assignment are available for a correctly-working implementation of the requirements specified in part 1 and 20% for each of parts 2 and 3. (If any part of your code fails to compile no marks will be available for attempts at parts 2 and 3.) 10% will be awarded for programming style and documentation and the remaining 5% for efficiency. (If you do not attempt all 3 parts the maximum mark available for style, documentation and efficiency will be proportional to the amount of the assignment that has been attempted.)

Submission

If all of your classes are in a single .java single file you should just submit this file to FASER. If your classes are in separate files you should submit the `.java` files inside a `.zip`, `.7z`, `.rar` or Linux `.tgz` file. You must not submit any class with a `main` method or any `.class` files.