Using Tries

By **luison9999**– *TopCoder Member*

Discuss this article

Introduction

There are many algorithms and data structures to index and search strings inside a text, some of them are included in the standard libraries, but not all of them; the trie data structure is a good example of one that isn't.

Let **word** be a single string and let **dictionary** be a large set of words. If we have a dictionary, and we need to know if a single word is inside of the dictionary the tries are a data structure that can help us. But you may be asking yourself, "Why use **tries** if **set** <**string**> and hash tables can do the same?" There are two main reasons:

> The tries can insert and find strings in O(*L*) time (where *L* represent the length of a single word). This is much faster than set , but is it a bit faster than a hash table.
>
> The **set** <**string**> and the hash tables can only find in a dictionary words that match exactly with the single word that we are finding; the trie allow us to find words that have a single character different, a prefix in common, a character missing, etc.

The tries can be useful in TopCoder problems, but also have a great amount of applications in software engineering. For example, consider a web browser. Do you know how the web browser can auto complete your text or show you many possibilities of the text that you could be writing? Yes, with the trie you can do it very fast. Do you know how an orthographic corrector can check that every word that you type is in a dictionary? Again a trie. You can also use a trie for suggested *corrections* of the words that are present in the text but not in the dictionary.

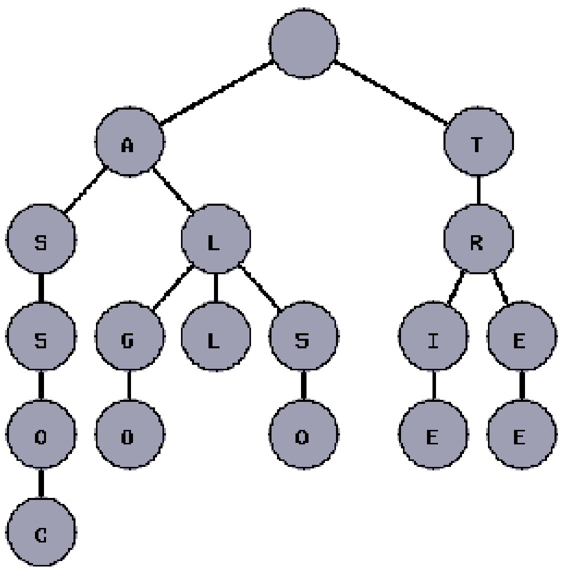
What is a Tree?

You may read about how wonderful the tries are, but maybe you don't know yet what the tries are and why the tries have this name. The word trie is an infix of the word "re**trie**val" because the trie can find a single word in a dictionary with only a prefix of the word. The main idea of the trie data structure consists of the following parts:

> The trie is a tree where each vertex represents a single word or a prefix.

> The root represents an empty string (“”), the vertexes that are direct sons of the root represent prefixes of length 1, the vertexes that are 2 edges of distance from the root represent prefixes of length 2, the vertexes that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that are *k* edges of distance of the root have an associated prefix of length *k*.
>
> Let **v** and **w** be two vertexes of the trie, and assume that **v** is a direct father of **w**, then **v** must have an associated prefix of **w**.

The next figure shows a trie with the words “tree”, “trie”, “algo”, “assoc”, “all”, and “also.”



Note that every vertex of the tree does not store entire prefixes or entire words. The idea is that the program should remember the word that represents each vertex while lower in the tree.

Coding a Trie

The tries can be implemented in many ways, some of them can be used to find a set of words in the dictionary where every word can be a little different than the target word, and other implementations of the tries can provide us with only words that match exactly with the target word. The implementation of the trie that will be exposed here will consist only of finding words that match exactly and counting the words that have some prefix. This implementation will be pseudo code because different coders can use different programming languages.

We will code these 4 functions:

> addWord. This function will add a single string **word** to the dictionary.
>
> countPreffixes. This function will count the number of words in the dictionary that have a string **prefix** as prefix.
>
> countWords. This function will count the number of words in the dictionary that match exactly with a given string **word**.

Our trie will only support letters of the English alphabet.

We need to also code a structure with some fields that indicate the values stored in each vertex. As we want to know the number of words that match with a given string, every vertex should have a field to indicate that this vertex represents a complete word or only a prefix (for simplicity, a complete word is considered also a prefix) and how many words in the dictionary are represented by that prefix (there can be repeated words in the dictionary). This task can be done with only one integer field words.

Because we want to know the number of words that have like prefix a given string, we need another integer field **prefixes** that indicates how many words have the prefix of the vertex. Also, each vertex must have references to all his possible sons (26 references). Knowing all these details, our structure should have the following members:

```
structure Trie
    integer words;
    integer prefixes;
    reference edges[26];
```

And we also need the following functions:

```
initialize(vertex)
addWord(vertex, word);
integer countPrefixes(vertex, prefix);
integer countWords(vertex, word);
```

First of all, we have to initialize the vertexes with the following function:

```
initialize(vertex)
    vertex.words=0

    vertex.prefixes=0
    for i=0 to 26
        edges[i]=NoEdge
```

The addWord function consists of two parameters, the vertex of the insertion and the word that will be added. The idea is that when a string **word** is added to a vertex **vertex**, we will add **word** to the corresponding branch of **vertex** cutting the leftmost character of word. If the needed branch does not exist, we will have to create it. All the TopCoder languages can simulate the process of cutting a character in constant time instead of creating a copy of the original string or moving the other characters.

```
addWord(vertex, word)
    if isEmpty(word)
        vertex.words=vertex.words+1
    else
        vertex.prefixes=vertex.prefixes+1
        k=firstCharacter(word)
        if(notExists(edges[k]))
            edges[k]=createEdge()
            initialize(edges[k])
        cutLeftmostCharacter(word)
        addWord(edges[k], word)
```

The functions countWords and countPrefixes are very similar. If we are finding an empty string we only have to return the number of words/prefixes associated with the vertex. If we are finding a non-empty string, we should to find in the corresponding branch of the tree, but if the branch does not exist, we have to return 0.

```
countWords(vertex, word)
    k=firstCharacter(word)
    if isEmpty(word)
        return vertex.words
    else if notExists(edges[k])
        return 0
    else
        cutLeftmostCharacter(word)
        return countWords(edges[k], word);

countPrefixes(vertex, prefix)
    k=firstCharacter(prefix)
    if isEmpty(word)
        return vertex.prefixes
    else if notExists(edges[k])
        return 0
    else
        cutLeftmostCharacter(prefix)
        return countWords(edges[k], prefix)
```

Complexity Analysis

In the introduction you may read that the complexity of finding and inserting a trie is linear, but we have not done the analysis yet. In the insertion and finding notice that lowering a single level in the tree is done in constant time, and every time that the program lowers a single level in the tree, a single character is cut from the string; we can conclude that every function lowers *L* levels on the tree and every time that the function lowers a level on the tree, it is done in constant time, then the insertion and finding of a word in a trie can be done in O(*L*) time. The memory used in the tries depends on the methods to store the edges and how many words have prefixes in common.

Other Kinds of Tries

We used the tries to store words with lowercase letters, but the tries can be used to store many other things. We can use bits or bytes instead of lowercase letters and every data type can be stored in the tree, not only strings. Let flow your imagination using tries! For example, suppose that you want to find a word in a dictionary but a single letter was deleted from the word. You can modify the countWords function:

```
countWords(vertex, word, missingLetters)
    k=firstCharacter(word)
    if isEmpty(word)
        return vertex.word
    else if notExists(edges[k]) and missingLetters=0
        return 0
    else if notExists(edges[k])
        cutLeftmostCharacter(word)
        return countWords(vertex, word, missingLetters-1)
        Here we cut a character but we don't go lower in the tree
    else
        We are adding the two possibilities: the first
        character has been deleted plus the first character is present
        r=countWords(vertex, word, missingLetters-1)

        cutLeftmostCharacter(word)
        r=r+countWords(edges[k], word, missingLetters)
        return r
```

The complexity of this function may be larger than the original, but it is faster than checking all the subsets of characters of a word.

Problems to Practice

WordFind SRM 232
SearchBox SRM 361
CyclicWords SRM 358
TagalogDictionary SRM 342
JoinedString SRM 302
CmpdWords SRM 268
Scruffle 2007 TCCC Marathon Online Round 1