Trie

Topics

1. Introduction to Trie
2. Count Prefix And Count Word Problem
3. Complexity Analysis
4. Problems
5. References

Introduction

Trie is an efficient information retrieval data structure. Using trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to M * log N, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in O(M) time.

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 present in the dictionary, trie is the data structure that can help us. But you may be asking yourself, "Why use tries if hash tables can do the same?" There are two main reasons:

1. The tries can insert and find strings in O(L) time (where L represent the length of a single word). This is a bit faster than a hash table.
2. 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.

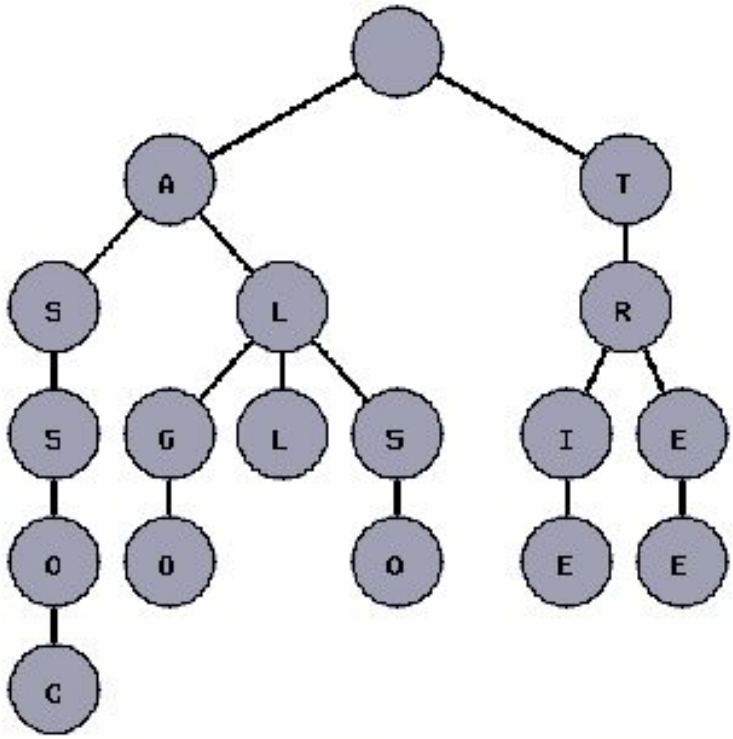
For example, consider a web browser. Do you know how the web browser can auto complete your text or show you many possibilities of the text that you could be writing? Yes, with the trie you can do it very fast. Do you know how an orthographic corrector can check that every word that you type is in a 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 Trie?

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:

1. The trie is a tree where each vertex represents a single word or a prefix.
2. 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*.
3. 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 following 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.

Implementation of 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.

Design Data Structure with following operations:

- **addWord( ) :** This function will add a single string word to the dictionary.
- **countPrefixes( ) :** 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**.

Solution using Trie Data Structure

```
// Asumming that our trie will take input in small letters [a-z]
#include<iostream>
#include<cstdio>
#include<cstring>
#include<cstdlib>
using namespace std;

// Assuming alhphabet is [a-z]
struct trienode {
   int words;//# of words
   int prefixes;//# of words having this prefix
   struct trienode * ref[26];//all possible refrences
};

// Creating Root Node
struct trienode* initialize() {
   struct trienode *p;
   p=(struct trienode*)malloc(sizeof(struct trienode));
   p->words=0;
   p->prefixes=0;
   for(int i=0; i<26; i++)
      p->ref[i]=NULL;
   return p;
}

// Adding Words
void addwords(struct trienode *root,char *word,int k,int wordlen) {
   if(k==wordlen)
      root->words++;
   else {
      int temp=word[k];
      temp-=97;
      if(root->ref[temp]==NULL) {
         root->ref[temp]=initialize();
      }
      root->ref[temp]->prefixes++;
      addwords(root->ref[temp], word, k+1, wordlen);
   }
}
```

```
// Counting Given Word
int countwords(struct trienode *root,char *word,int k,int wordlen) {
    if(k == wordlen)
        return root->words;

    int temp=word[k];
    temp-=97;
    if(root->ref[temp]==NULL)
        return 0;
    else
        return countwords(root->ref[temp], word, k+1, wordlen);
}

// Counting Words Having Same Prefix
int countprefixes(struct trienode *root,char *prefix,int k,int prefixlen) {
    if(k==prefixlen)
        return root->prefixes;

    int temp=prefix[k];
    temp-=97;
    if(root->ref[temp]==NULL)
        return 0;
    else
        return countprefixes(root->ref[temp],prefix,k+1,prefixlen);
}

int main() {
    struct trienode *root;
    root=initialize();

    addwords(root,"tree",0,4);
    addwords(root,"treek",0,5);
    addwords(root,"trie",0,4);
    addwords(root,"assoc",0,5);
    addwords(root,"all",0,3);
    addwords(root,"algo",0,4);
    addwords(root,"also",0,4);

    cout<<countprefixes(root,"tr",0,2)<<endl;
    cout<<countprefixes(root,"al",0,2)<<endl;

    cout<<countwords(root,"tree",0,4)<<endl;
    cout<<countwords(root,"also",0,4)<<endl;

    return 0;
}
```

Time Complexity
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.

Space Complexity
The memory used in the tries depends on the methods to store the edges and how many words have prefixes in common. But we can calculate the upper bound of memory.
Let's Alphabet size is A and maximum length of word is W.
Space Sum = $A + A^2 + A^3 + A^4 + \ldots\ldots\ldots\ldots + A^I = A(A^{(I+1)}-1)/(A-1)$
So, we can say that Space Complexity is **$O(A^I)$**

Problems

1. Design a spell-checker
   Solution: Build Trie of all words and perform searching

2. Build Auto-complete/word suggestion
   Solution: Build Trie of all words and during search populate all words or (K words) as suggestion.

3. Given a list of phone numbers, determine if it is consistent in the sense that no number is the prefix of another.

   Example: A:911, B:97625999, C:91125426
   In this case, it's not possible to call C, because the central would direct your call to the A as soon as you had dial the first three digits of C's phone number. So this list would not be consistent.
   Solution: Create a trie with alphabet [0-9] and insert phone numbers in the trie. During insertion check if there is already any word present in the trie which is prefix of current phone number. For identifying nodes which represent end of word, you can maintain a flag (boolean) in the node structure of the trie.

4. Given an array of integers, we have to find two elements whose XOR is maximum.
   Solution: https://threads-iiith.quora.com/Tutorial-on-Trie-and-example-problems
   a. Create trie using bit representation of all the numbers, starting with the MSB bit. Observer that, the MSB bits contributes more.
   b. Now for any element ar[i], find its correspoding pair which gives maximum xor value for this element using trie.

References

1. http://www.geeksforgeeks.org/trie-insert-and-search/
2. https://www.cs.bu.edu/teaching/c/tree/trie/
3. http://www.geeksforgeeks.org/trie-delete/