N-ary Tree - Trie

Topics

1. Introduction to Trie
2. What do we mean by ‘Trie’?
3. Implementation of Trie
4. Time Complexity
5. Space Complexity
6. Problems & Solutions
7. Important 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 a binary search tree, a well balanced BST will need time proportional to $M \times \log_2(N)$, where M is maximum string length and N is number of keys in the tree. Using trie, we can search for 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 represents the length of a single word). This is a bit faster than a hash table.
2. Hash tables can only find for the word in a dictionary of words that match exactly with the single word that we are trying to find; however the trie allows us to find words that have a single character different, a prefix in common, a character missing, etc.

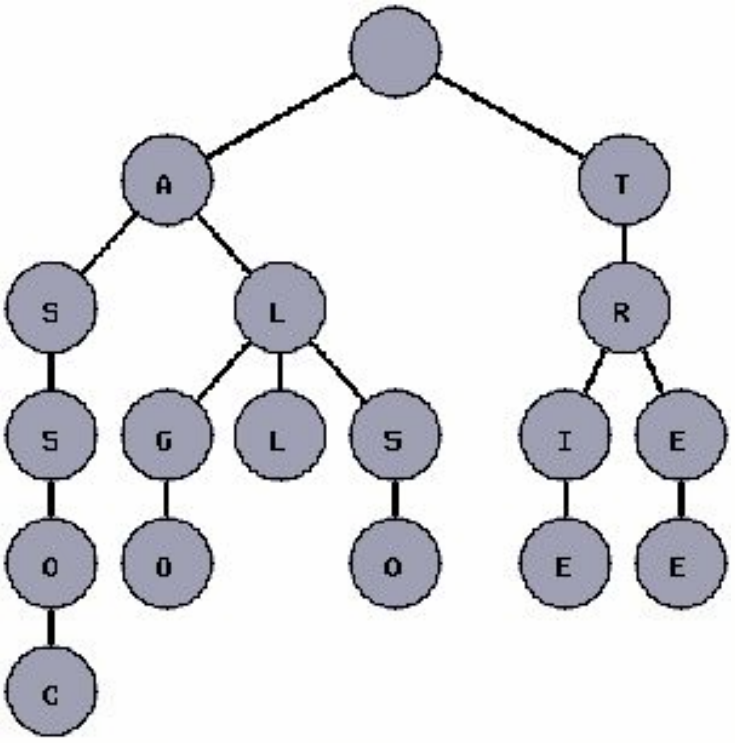
For example, consider a web browser. Do you know how the web browser can auto complete your text or show you many possibilities of the text that you could be writing? Yes, with the help of trie you can do it in a very quick method. Do you know how an orthographic corrector can check that every word that you type is in a dictionary? Again a trie. You can also use a trie for suggestions on corrections of the words that are present in the text but not in the dictionary.

What do we mean by '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 vertices that are direct sons of the root represent prefixes of length 1, the vertices that are 2 edges of distance from the root represent prefixes of length 2, the vertices that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that is *k* edges of distance from the root has an associated prefix of length *k*.
3. Let **v** and **w** be two vertices 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 iterating over the trie.

Implementation of Trie

A trie 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 trie 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 with a given query and also counting the words that start with a given prefix.

Design a 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 start with a given **prefix**.
- **`countWords()`:** This function will count the number of words in the dictionary that match exactly with a given string (word).

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

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

/* 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;

        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;
    }
```

- **Implementation of Trie in Java**
- **Implementation of Trie in Python**

Time Complexity

In the introduction you might've read that the complexity of finding and inserting an element into a trie is linear, but we have not done the analysis yet. When we insert of find an element, notice that lowering a single level in the trie is done in constant time, and every time that the program lowers a single level in the trie, a single character is cut from the string; we can conclude that every function lowers **L** levels on the trie and every time that the function lowers a level on the trie, it is done in constant time. So the complexity of insert and find operations of a word in a trie is **O(L)** time, where **L** denotes length of the word.

Space Complexity

The memory used in the trie 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 assume *alphabet size* is **A** and *maximum length of word* is **W**.

Space Sum = $A + A^2 + A^3 + A^4 + \ldots\ldots\ldots\ldots + A^I = \frac{A\,(A^{(I+1)} - 1)}{A - 1}$

So, we can say that Space Complexity is $\mathbf{O(A^I)}$

However, if we take into account that we are inserting **N** words, where *maximum length of a word* is **W**, we have **NxW** characters, and each of these characters will result in creating a new node in the trie (worst case).

Problems & Solutions

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 A as soon as you had dialed the first three digits of C's phone number. So this list would be inconsistent.
   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: Threads@IIITH-Blog-QuoraReference
   a. Create trie using bit representation of all the numbers, starting with the MSB bit. Observe that, the MSB bits contributes most.
   b. Now for any element `ar[i]`, find its corresponding pair which gives maximum XOR value for this element using trie.

Important References

1. **Trie | Insert & Search --- GeeksForGeeksReference**
2. **Trie | Delete --- GeeksForGeeksReference**
3. **More on Trie @BostonUniversity**