GİRİŞ

Bir metnin içinde bir stringi aramak için birçok algoritma ve veri yapısı vardır. Bunların bir kısmı standart kütüphanelerde yer alırken, diğerleri standard kütüphanelerde bulunmaz; Trie veri yapısı standart kütüphanede yer almayanlara güzel bir örnektir.

Eğer bir sözlüğümüz varsa ve bir kelimenin bu sözlükte yer alıp almadığını bilmek istiyorsak trie veri yapısı bunu yapmak için çok uygundur. Belki şu soruyu sorabilirsiniz, "**set<string>** ve **hash tabloları** ile aynı şeyi yapabiliyorken neden **trie** kullanalım?" Bunun için temel olarak iki sebep var:

Trieler bir stringi ekleme veya bulma işlemini O(L) zamanda yapabilirler (L burada stringin uzunluğunu temsil ediyor). Bu setten çok daha hızlıyken, hash tablolarından sadece biraz hızlıdır.

Set<string> ve hash tabloları ile yapabildiğimiz tek şey bir stringin bir sözlükte eşleştiği kelimeyi bulmaktır. Fakat, trie sadece bir karakterleri farklı olan, ortak prefixleri olan,vb. stringleri bulabilmemize de olanak sağlar.

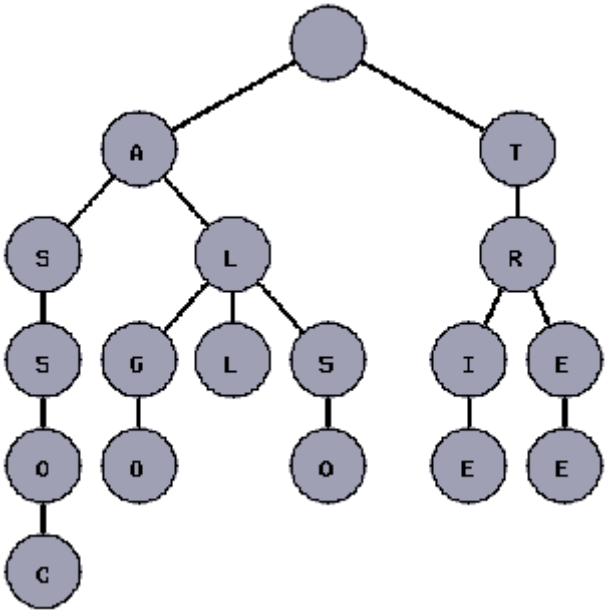
Trielerin Topcoder, Codeforces gibi yarışma sitelerdeki problemleri çözmede işe yaramasının yanında, bilgisayar mühendisliğinde de birçok uygulamaları vardır. Mesela bir web browserı düşünün. Bir web browserın sizin yazdığınız metni nasıl otomatik tamamladığını veya sizin yazmakta olabileceğiniz bir sürü muhtemel kelimeyi nasıl gösterdiğini biliyor muydunuz? Bu işlem trieyle çok hızlı bir şekilde yapılabilir. Bir yazım hatası düzelticinin yazdığınız her kelimenin sözlükte olup olmadığını nasıl kontrol ettiğini biliyor muydunuz? Bunun için de yine trie kullanılır.

TRIE NEDİR?

Trielerin çok işe yaradığını duymuş olabilirsiniz, ama belki de trielerin ne olduğunu ve neden böyle isimlendirildiğini bilmiyorsunuz. Trie kelimesi re**trie**val(bulup getirme) kelimesinin bir infixidir. Trie veri yapısının ana mantığı şunları içerir:

- Trie her nodun bir kelimeyi veya bir kelimenin prefixini gösterdiği bir ağaçtır.
- Ağacın kökü boş bir stringi temsil eder (""). Kökün ilk çocukları 1 uzunluğundaki prefixleri, köke 2 edge uzaklıktaki nodlar 2 uzunluğundaki prefixleri, 3 edge uzaklıktakiler 3 uzunluğundaki prefixleri … temsil eder. Yani, köke k edge uzaklıktaki bir nod k uzunluğunda bir prefixi temsil eder.
- **v** ve **w** ağacın iki nodu olduğunu ve v'nin w'nun direkt atası olduğunu varsayalım. O zaman v'nin belirttiği string w'nun belirtiğinin bir prefixidir.

Aşağıdaki şekil şu kelimelerle oluşan trie yapısını göserir: "tree","trie","algo","assoc","all" ve "also".



TRIEYI KODLAMA

Trie kullanılma amacına göre birçok şekilde kodlanabilir. Burada kodu gösterilecek olan trie bir stringin bir sözlükte yer alıp almadığını bulmaya ve bir sözlükte belli bir prefixe sahip kaç kelime olduğunu saymaya izin verir. Triemizin sadece ingilizcedeki küçük harfleri içerdiğini varsayıyoruz.

Toplamda 4 fonksiyon kodlayacağız:

initialize

Bu fonksiyon yeni bir nodu trieye eklemek için kullanılır.

addWord

Bu fonksiyon verilen bir kelimeyi sözlüğümüze ekler.

countPreffixes

Bu fonksiyon verilen bir stringin sözlüğümüzdeki kaç kelimenin prefixi olduğunu bulur.

countWords

Bu fonksiyon verilen bir stringin sözlükte kaç kere geçtiğini bulur.

Ek olarak, her bir nod için şu bilgileri tutmamız gerekir:

- Sözlükte bu nodun belirttiği stringle aynı kaç kelime vardır
- Sözlükte prefixi bu nodun belirttiği string olan kaç kelime vardır
- Bu noddan çıkabilecek 26 muhtemel kenarın bilgileri

Yani triedeki nodların yapısı şöyledir:

```
struct Nod{
        int word,prefix;
        int edge[26];
}
// triedeki tüm nodları içeren dizi
Nod nodlar[MAX_N];
// triedeki nod sayısını temsil eder ve nod ekledikçe artar
int sz=0;
```

initialize fonksiyonumuz şöyledir:

```
void initialize(){

        sz++;
        nodlar[sz].word = 0;
        nodlar[sz].prefix = 0;
        for(i,0,26)     nodlar[sz].edge[i] = 0;

}
```

initialize fonksiyonu en başta root nodu oluşturmak için 1 kere çağırılır. Rootun numarası 1olur. Yani **nodlar** dizisinin 1.indisinde root nodun bilgileri vardır.

addWords fonksiyonumuz şöyledir:

```
void addWords( string word ){

        int pos=1;

        for(i,0,word.size()){

                char c = word[i] - 'a';
                if( nodlar[pos].edge[c] == 0 ){

                        initialize();
                        nodlar[pos].edge[c] = sz;
                        pos = sz;

                }
                else
                        pos = nodlar[pos].edge[c];

                nodlar[pos].prefix++;

        }

        nodlar[pos].word++;

}
```

countWords fonksiyonumuz söyledir:

```
int countWords(string word){

        int pos = 1;
```

```
        for(i,0,word.size()){

                char c = word[i] - 'a';

                if( nodlar[pos].edge[c] != 0 )
                        pos = nodlar[pos].edge[c];
                else
                        return 0;
        }

        return nodlar[pos].word;

}
```

countPrefixes fonksiyonumuz söyledir:

```
int countPrefixes(string word){

        int pos = 1;

        for(i,0,word.size()){

                char c = word[i] - 'a';

                if( nodlar[pos].edge[c] != 0 )
                        pos = nodlar[pos].edge[c];
                else
                        return 0;
        }

        return nodlar[pos].prefix;

}
```

KARMAŞIKLIK ANALİZİ

addWords, countWords ve countPrefixes fonksiyonları parametre olarak gelen stringiniz uzunluğu kadar işlem yapar. Bu yüzden toplam maliyetimiz sözlükteki stringlerin ve sorgulanan stringlerin hepsinin uzunlukları toplamı kadardır.