T9 Dictionary with Trie Data Structure

Shashank Kumar Tekriwal (72672)

B.Tech.(IT and Mathematical Innovations)
Cluster Innovation Centre
University of Delhi

shashank.internet@gmail.com

***Abstract*—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. However the penalty is on trie storage requirements.**

I. Introduction

T-9, which stands for Text on 9 keys, is a USA-patented predictive text technology (1) for mobile phones (specifically those that contain a 3x4 numeric keypad), originally developed by Tegic Communications, now part of Nuance Communications.

T9's objective is to make it easier to type text messages. It allows words to be entered by a single key press for each letter, as opposed to the multi-tap approach used in conventional mobile phone text entry, in which several letters are associated with each key, and selecting one letter often requires multiple key presses.

It combines the groups of letters on each phone key with a fast-access dictionary of words. It looks up in the dictionary all words corresponding to the sequence of key presses and orders them by frequency of use.

.

A. Implementation

Implementing a T9 dictionary would require very fast text searches for words in a dictionary. *Trie* comes from the word *re**trie**val*. It ensures an optimal time complexity for *insert* and *search* operations.

II. Trie not a tree

A trie is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node, instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.

A trie can be understood as a binary tree but unlike binary trees that can have two branches, a trie can have multiple branches. Also, in a trie the values in the node is associated with the position of the node and doesn't depend on the value of its parent or child node. As the values in the nodes are static, tries have an advantage over traversal, insertion and searching time complexities.

A general trie would look somewhat like:

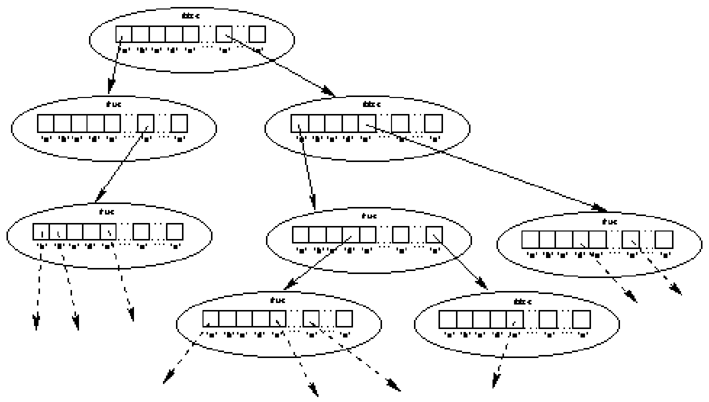

Figure 1: Conceptual image of trie

A. Prefix Tree

Trie data structure is also called a *prefix tree* (2) as it stores the prefixes of strings only once. New strings are appended to the existing prefixes already in the tree. For example: If my trie has the word 'go' in it and I insert another word 'good' into it, only 2 new nodes are formed- '*o* and *d*'. A typical trie with the following strings would look like:

Strings: *car, card, cart, cat, cel, celery, close, closet, closely, clue*

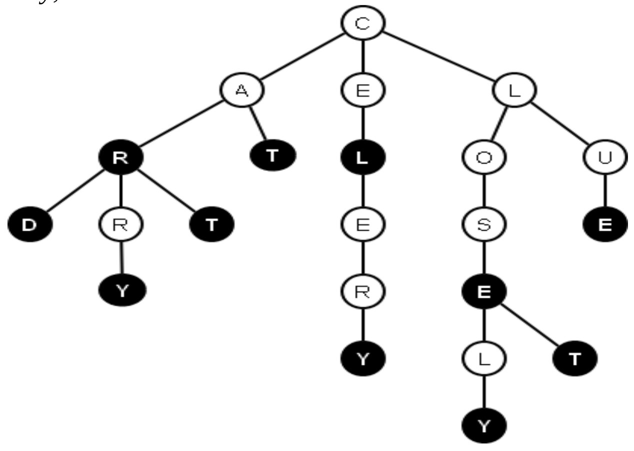

Figure 2: Example of Trie

The black node represent a flag for the end-of-word. If we search for the word *'cat'* in the above trie it would return *true*, but a search for *'ca'* false.

Thus, from the above the structure of trie is quite clear. It is quite intuitive and interesting.

III. Trie Implementation

I have written a Java implementation of the Trie data structure. It is extremely fast. It took only 2.5 seconds to build a tree with as many as 235,886 words – a full English dictionary.

A. *Trie Node*

The Trie Node would look something like:

```
public class TrieNode{
        int colour;
        char letter;
        TrieNode child[];

        public TrieNode(int colour, char letter, int len){
                this.colour = colour;
                this.letter = letter;
                this.child = new TrieNode[len];

        }
}
```

The above code is self-explanatory. It forms the base of the trie data structure.

TrieNode itself contains an array of trieNode[] type which are its multiple branches.

Colour attribute defines a flag for the end of word.

Letter defines the characted stored in the node.

Len defines the maximum number of branches the node can have.

B. *Insertion in Trie*

The insertion process in a trie takes *O(L)* time where L is the length of the string to be inserted. It is much faster than a well balanced binary tree that would take *O(M*log N)* where N is the number of nodes. It can be bit faster than an imperfect hash table as it may contain collisions. The worst case complexity of an imperfect hast table would be *O(N)* whereas worst case complexity of trie still remains *O(L).*

The insert function would look something like:

```
public void insert(String str){
        str = str.toLowerCase();
        int length = str.length();
        TrieNode temp = root;
        for (int x = 0; x < length; x++){
                char ch = str.charAt(x);
             TrieNode temp2 = temp.getChild((int)ch-97);
                if(temp2==null){
                        temp.setChild(ch,26);
                temp2 = temp.getChild((int)ch-97);
                }
                if(x==length-1){
                        temp2.setColour(1);
                }
                temp = temp2;
        }
}
```

Thus, the insertion subroutine runs in *O(L)* time.

The value of the colour attribute is changed as soon as the word completes.

C. *Searching in Trie*

The search function returns a boolean value telling whether the string being searched was found or not.

The search function would look like:

```
public boolean search(String str){
        str = str.toLowerCase();
        String pattern = "^[a-zA-Z]+$";
        if(!str.matches(pattern)){
                return false;
        }
        int length = str.length();
        TrieNode temp = root;
        for(int x = 0; x<length; x++){
                char ch = str.charAt(x);
                temp = temp.getChild((int)ch-97);
                if(temp==null || (x==length-1 &&
temp.getColour()==0)){
                        return false;
                }
        }
        return true;
}
```

I have used regular expression to filter strictly only alphabetic words.

We see that the above code has a efficient time complexity of O(L). Again, all comparisons with BST and Hash Tables are valid in this case too.

IV. Disadvantages

Even though trie is a very text searching data structure it is quite unpopular.

One major disadvantage of a trie is its space complexity. The memory requirements of trie is O(ALPHABET_SIZE * key_length * N) where N is number of keys in trie.

My dictionary of English words took 2.5 KB of space on disk. The corresponding trie took around 30 KB when I build the trie and *serialized* it to disk.

More popular data structres like Hash Maps can be used as an alternative to trie. Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random access time is high compared to main memory. Some keys, such as floating point numbers, can lead to long chains and prefixes that are not particularly meaningful. Nevertheless a bitwise trie can handle standard IEEE single and double format floating point numbers.

V. T9 Implementation

Among all Data Structures, using Trie is the most optimal solution (3) for the T9 implementation for various reasons:

- Tries support **ordered iteration**, whereas iteration over a hash table will result in a pseudorandom order given by the hash function (also, the order of hash collisions is implementation defined), which is usually meaningless
- **Tries facilitate longest-prefix matching, but hashing does not, as a consequence of the above. Performing such a "closest fit" find can, depending on implementation, be as quick as an exact find.**
- Tries tend to be faster on average at insertion than hash tables because hash tables must rebuild their index when it becomes full - a very expensive operation. Tries therefore have **much better bounded worst case time costs**, which is important for latency sensitive programs running on server.

The purpose of the T9 dictionary is to predict text on the input of a few prefixes. For an example, if the user gives *dict* as the input – the corresponding output should suggest *dictate or dictation*. With a trie implementation it is quite easy to follow the prefixes upto 1 or more depths to find words that are close to the prefix input.

The complexity of the above extened search would remain *s*O(L)* where *s* is the number of words the script finds.

A. Permutations of Input

1
2 abc
3 def
4 ghi
5 jkl
6 mno
7 pqrs
8 tuv
9 wxyz
*+
0
#

Each button on the mobile pad corresponds to 3 letters. This inceases the complexity of the algorithm. For each input the number of possible combinations get tripled.

With an input of 4 buttons, it is 3^4 permutations that needs to be first computed and then checked for the their validity.

All these permuted strings are first inserted into a Queue, and then searched on the trie. All resulting solutions then can be added to another Queue and displayed as an output.

B. Priority Setting

Priority setting (1) is done in a t9 dictonary to filter the search results. A lot of prefixes are formed on the t9 num pad and so correspondingly there are quite a huge number of search results obtained which makes the system complicated. While creating the trie, we can also set priority of words by storing some value in the node.

The priority would depend upon the frequency of the words used in English. While looping through the results we can produce results that have higher priority as compared to the ones with lower.

This would help in narrowing down the choices for the user and make the system more user friendly.

VI. Conclusion

Trie data structures have their unique properties that are incomparable. Advanced variations of trie include Compact Prefix trie, Radix Trie, Suffix Tree etc. that are more space compact.

This was very exciting project. I could have put more efforts if time had permitted. I got to learn a lot about various Data Structures – Hash maps, BST, Heaps etc. through the course of my poject.

Acknowledgment

I express my warm thanks to Mr. Alok Nikhil Jha for assigning me this project. I am thankful for his aspiring guidance, invaluably constructive criticism and friendy advice during the project work.

REFERENCES

1. Trie - Wikipedia. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Trie.
2. Tries. *cs.lmu.* [Online] http://cs.lmu.edu/~ray/notes/tries/.
3. Data Structure Trie. *blogsopt.in.* [Online] http://jayant7k.blogspot.in/2011/06/data-structures-trie.html.