208. Implement Trie (Prefix Tree)

Summary

This article is for intermediate level users. It introduces the following ideas: The data structure Trie (Prefix tree) and most common operations with it.

Solution

Applications

Trie (we pronounce "try") or prefix tree is a tree data structure, which is used for retrieval of a key in a dataset of strings. There are various applications of this very efficient data structure such as :

1. Autocomplete

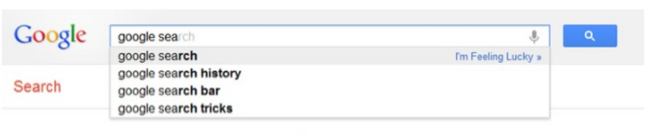

Figure 1. Google Suggest in action.

2. Spell checker

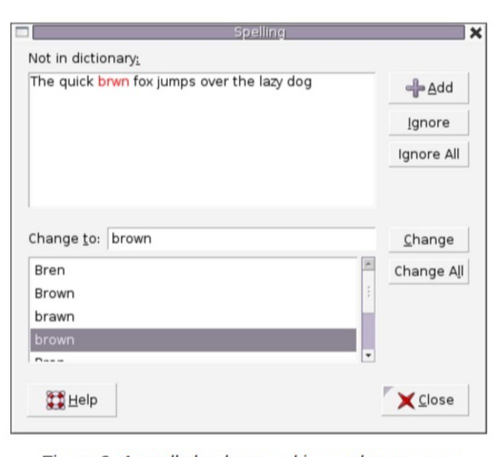

Figure 2. A spell checker used in word processor.

3. IP routing (Longest prefix matching)

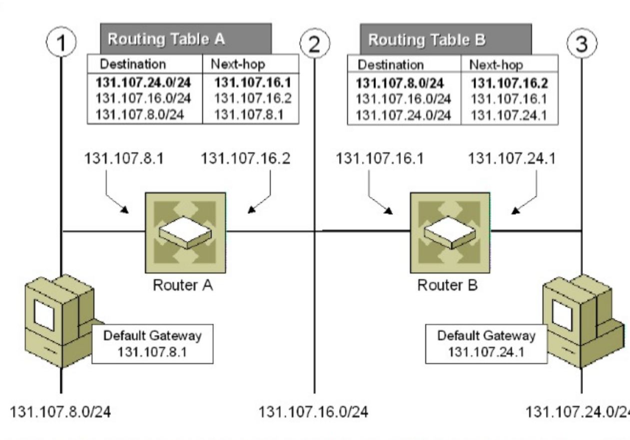

Figure 3. Longest prefix matching algorithm uses Tries in Internet Protocol (IP) routing to select an entry from a forwarding table.

4. T9 predictive text

Figure 4. T9 which stands for Text on 9 keys, was used on phones to input texts during the late 1990s.

5. Solving word games

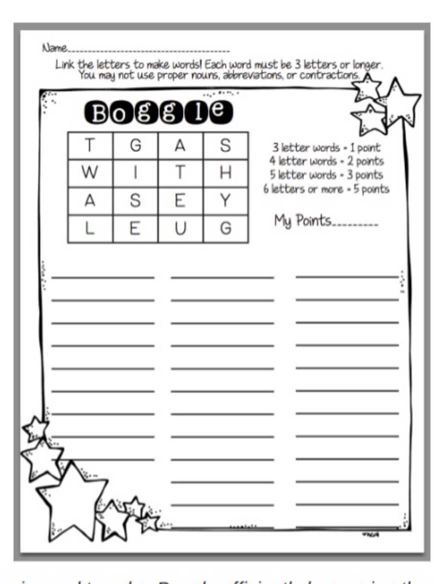

Figure 5. Tries is used to solve Boggle efficiently by pruning the search space.

There are several other data structures, like balanced trees and hash tables, which give us the possibility to search for a word in a dataset of strings. Then why do we need trie? Although hash table has $O(1)$ time complexity for looking for a key, it is not efficient in the following operations :

- Finding all keys with a common prefix.
- Enumerating a dataset of strings in lexicographical order.

Another reason why trie outperforms hash table, is that as hash table increases in size, there are lots of hash collisions and the search time complexity could deteriorate to $O(n)$, where n is the number of keys inserted. Trie could use less space compared to Hash Table when storing many keys with the same prefix. In this case using trie has only $O(m)$ time complexity, where m is the key length. Searching for a key in a balanced tree costs $O(m \log n)$ time complexity.

Trie node structure

Trie is a rooted tree. Its nodes have the following fields:

- Maximum of R links to its children, where each link corresponds to one of R character values from dataset alphabet. In this article we assume that R is 26, the number of lowercase latin letters.
- Boolean field which specifies whether the node corresponds to the end of the key, or is just a key prefix.

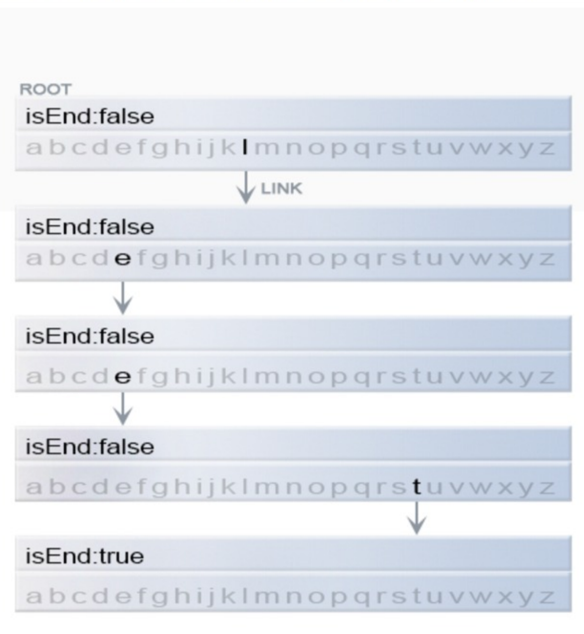

Figure 6. Representation of a key "leet" in trie.

Java

```
class TrieNode {

    // R links to node children
    private TrieNode[] links;

    private final int R = 26;

    private boolean isEnd;

    public TrieNode() {
        links = new TrieNode[R];
    }

    public boolean containsKey(char ch) {
        return links[ch -'a'] != null;
    }
    public TrieNode get(char ch) {
        return links[ch -'a'];
    }
    public void put(char ch, TrieNode node) {
        links[ch -'a'] = node;
    }
    public void setEnd() {
        isEnd = true;
    }
    public boolean isEnd() {
        return isEnd;
    }
}
```

Two of the most common operations in a trie are insertion of a key and search for a key.

Insertion of a key to a trie

We insert a key by searching into the trie. We start from the root and search a link, which corresponds to the first key character. There are two cases :

- A link exists. Then we move down the tree following the link to the next child level. The algorithm continues with searching for the next key character.
- A link does not exist. Then we create a new node and link it with the parent's link matching the current key character. We repeat this step until we encounter the last character of the key, then we mark the current node as an end node and the algorithm finishes.

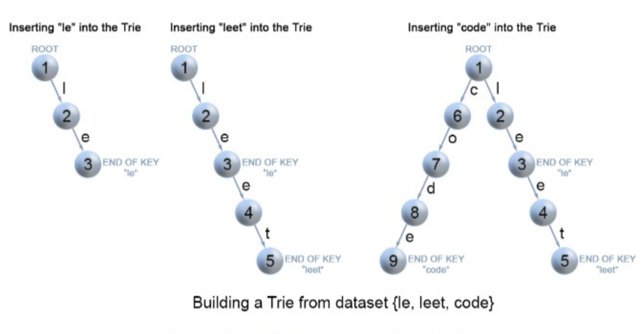

Figure 7. Insertion of keys into a trie.

Java

```
class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
            char currentChar = word.charAt(i);
            if (!node.containsKey(currentChar)) {
                node.put(currentChar, new TrieNode());
            }
            node = node.get(currentChar);
        }
        node.setEnd();
    }
}
```

Complexity Analysis

- Time complexity : $O(m)$, where m is the key length.

In each iteration of the algorithm, we either examine or create a node in the trie till we reach the end of the key. This takes only m operations.

- Space complexity : $O(m)$.

In the worst case newly inserted key doesn't share a prefix with the the keys already inserted in the trie. We have to add m new nodes, which takes us $O(m)$ space.

Search for a key in a trie

Each key is represented in the trie as a path from the root to the internal node or leaf. We start from the root with the first key character. We examine the current node for a link corresponding to the key character. There are two cases :

- A link exist. We move to the next node in the path following this link, and proceed searching for the next key character.
- A link does not exist. If there are no available key characters and current node is marked as `isEnd` we return true. Otherwise there are possible two cases in each of them we return false :
  - There are key characters left, but it is impossible to follow the key path in the trie, and the key is missing.
  - No key characters left, but current node is not marked as `isEnd`. Therefore the search key is only a prefix of another key in the trie.

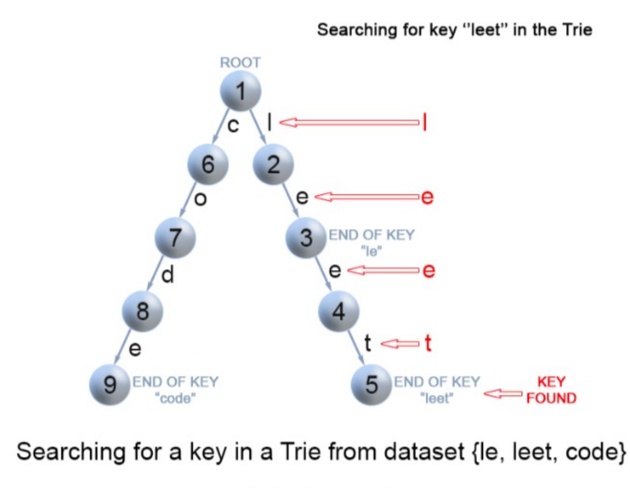

Figure 8. Search for a key in a trie.

Java

```
class Trie {
    ...

    // search a prefix or whole key in trie and
    // returns the node where search ends
    private TrieNode searchPrefix(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
           char curLetter = word.charAt(i);
           if (node.containsKey(curLetter)) {
               node = node.get(curLetter);
           } else {
               return null;
           }
        }
        return node;
    }

    // Returns if the word is in the trie.
    public boolean search(String word) {
       TrieNode node = searchPrefix(word);
       return node != null && node.isEnd();
    }
}
```

Complexity Analysis

- Time complexity : $O(m)$ In each step of the algorithm we search for the next key character. In the worst case the algorithm performs m operations.
- Space complexity : $O(1)$

Search for a key prefix in a trie

The approach is very similar to the one we used for searching a key in a trie. We traverse the trie from the root, till there are no characters left in key prefix or it is impossible to continue the path in the trie with the current key character. The only difference with the mentioned above `search for a key` algorithm is that when we come to an end of the key prefix, we always return true. We don't need to consider the `isEnd` mark of the current trie node, because we are searching for a prefix of a key, not for a whole key.

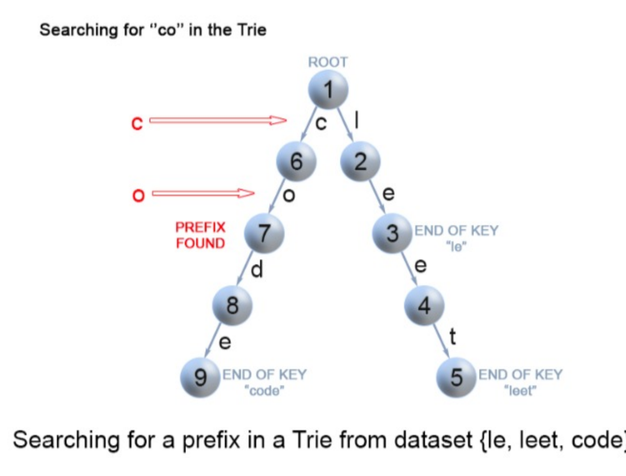

Figure 9. Search for a key prefix in a trie.

Java

```
class Trie {
    ...

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    public boolean startsWith(String prefix) {
        TrieNode node = searchPrefix(prefix);
        return node != null;
    }
}
```

Complexity Analysis

- Time complexity : $O(m)$
- Space complexity : $O(1)$

Practice Problems

Here are some wonderful problems for you to practice which uses the Trie data structure.

1. Add and Search Word - Data structure design - Pretty much a direct application of Trie.
2. Word Search II - Similar to Boggle.

Analysis written by: @elmirap.

Java textmate

```
public class Trie {

    private Trie[] branches;
    public int times;

    /** Initialize your data structure here. */
    public Trie() {
        branches = new Trie[26];
        times = 0; // Number of words ending here
    }

    /** Inserts a word into the trie. */
    public void insert(String word) {
        Trie curNode = this;
        for (int i = 0; i < word.length(); i++) {
            int idx = word.charAt(i) - 'a';
            if (curNode.branches[idx] != null) {
                curNode = curNode.branches[idx];
            } else {
                curNode.branches[idx] = new Trie();
                curNode = curNode.branches[idx];
            }
        }
        curNode.times += 1;
    }

    /** Returns if the word is in the trie. */
    public boolean search(String word) {
        Trie curNode = this;
        for (int i = 0; i < word.length(); i++) {
            int idx = word.charAt(i) - 'a';
            if (curNode.branches[idx] != null) {
                curNode = curNode.branches[idx];
            } else {
                return false;
            }
        }
        return curNode.times >= 1;
    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    public boolean startsWith(String prefix) {
        Trie curNode = this;
        for (int i = 0; i < prefix.length(); i++) {
            int idx = prefix.charAt(i) - 'a';
            if (curNode.branches[idx] == null) {
                return false;
            } else {
                curNode = curNode.branches[idx];
            }
        }
        return true;
    }
}

/**
 * Your Trie object will be instantiated and called as such:
 * Trie obj = new Trie();
 * obj.insert(word);
 * boolean param_2 = obj.search(word);
 * boolean param_3 = obj.startsWith(prefix);
 */
```

Custom Testcase (Contribute) Run Code Submit Solution

["Trie","insert","search","search","insert","search","insert","search"]
[[],["abc"],["abc"],["ab"],["ab"],["ab"],["ab"],["ab"]]

How to create a testcase

Submission Result: Accepted More Details

Next challenges: Add and Search Word - Data structure design

Share your acceptance!

4

Check out our solution! Reveal Solution