Tries

By Ralph McDougall

IOI Camp 3

4 March 2017

What is a trie?

- A trie is sometimes called a digital tree, radix tree or prefix tree
- Trie is pronounced either “tree” or “try”, depending on who you ask
- A trie is a data structure that stores words from a dictionary
- It allows O(L) insertion where L is the length of the word being added
- It allows O(L) existence detection where L is the length of the word being checked for
- Since it stores prefixes, it allows us to find all words in a dictionary with a certain prefix

Implementation

- To implement a trie, just like with a tree, we need some nodes
- Each node stores an array containing all of its children and whether or not it is an endpoint.
- We also need a root node which acts as the start point for the search and insert algorithms.

Implementation

```
#define ALPHABET_SIZE 26

struct Node
{
    struct Node* children[ALPHABET_SIZE];
    bool isEnd;
};
```

That’s all there is to it!

Some important things to include

```
int getIndex(char c)
{
    return c - 65; // If only uppercase letters are used
}

Node* getNode(void)
{
    Node *n = NULL;
    n = (struct Node*)malloc(sizeof(Node));
    if (n)
    {
        n->isEnd = false;
        for (int i = 0; i < ALPHABET_SIZE; i++)
        {
            n->children[i] = NULL;
        }
    }
    return n;
}

Node* root = getNode();
```

Insertion

- We start at the root vertex
- We go to the child vertex corresponding to the next character of the word that we are inserting
- If that node does not exist, we add a new node there and go to it
- We keep doing this until we have checked all of the characters
- We mark the last vertex visited as an end node for a word.
- Notice that adding a new node and moving to a new node takes $O(1)$ and checking each character takes $O(L)$
- Thus insertion is $O(L)$

Insertion

```
void insert(string s)
{
    Node* currNode = root;

    for (int i = 0; i < s.size(); ++i)
    {
        int index = getIndex(s[i]);
        if (!currNode->children[index])
        {
            currNode->children[index] = getNode();
        }
        currNode = currNode->children[index];
    }
    currNode->isEnd = true;
}
```

Search

- We start at the root vertex
- If the next character isn’t present, the word isn’t present, so we can return false
- We move to the next character and repeat this until we have checked all of the characters
- We return true if the last node is an end node and it is valid
- Checking if a character is present takes O(1) and checking each character takes O(L)
- Thus, searching takes O(L)

Search

```
bool search(string s)
{
    Node *currNode = root;
    for (int i = 0; i < s.size(); ++i)
    {
        int index = getIndex(s[i]);
        if (!currNode->children[index])
        {
            return false;
        }
        currNode = currNode->children[index];
    }
    return (currNode != NULL && currNode->isEnd);
}
```

Analysis

- Time complexity:
  - Insertion: O(L)
  - Search: O(L)
- Space complexity: O(NL)

Example Question

You are given a dictionary of N words and word W. You want to find the "closest match" to W in the dictionary. The "closest match" is the word that differs from W in the least number of characters.

("car" is closer to "cat" than "bat" is to "cat")

Bruce Force Solution

Generate each possible word that can be obtained from W by changing a few characters. This takes exponential time, which is terrible!

Trie Solution

We create a prefix tree of all of the words in the dictionary. We now do a Dijkstra on the tree keeping track of how many characters it differs by at each stage.

We can calculate the answer much faster using this.

Trie Solution

We can construct the prefix tree in O(LN) where L is the length of the longest word. We can check if a word is in the prefix tree in O(LK). Thus we can complete the problem in $O(L^2NK)$. Note that this is the worst case scenario.