COMP290: Trie Spell Checker

Thomas Chambers

Spell Checker

This project use's a C++ implementation of a Trie data structure to contain the entire English language. A search algorithm is used to implement a spell checker, allowing the user to check their spelling of words and to also add new words to the Trie.

This approach uses several optimizations to handle creation, insertion and search. I've also explored other possible implementations and weighed up their pros and cons.

To write a spell checker there are two large requirements

- It must be able to contain lots of data, as it needs to contain all the words in the English language
- It must have a low Big(0), as to be able to search for the word instantaneously and successfully.

What is a Trie

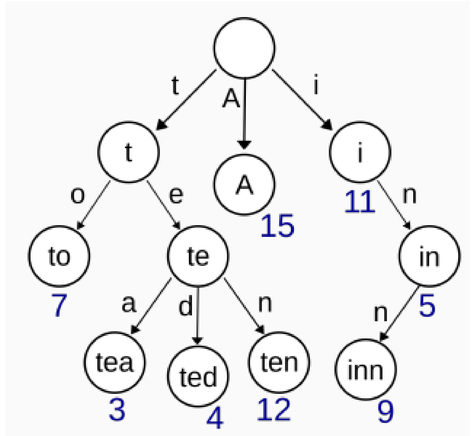

Figure 1. Example Trie Diagram, public domain

A Tree is a hierarchical data structure comprised of nodes containing information. Each node points to one or more node(s) below it in the structure, known as leaves.

A Trie is a form of tree used to find keys within a set. Unlike other Tree structures, such as a binary tree or AVL tree, a Trie does not need any balancing actions performed upon it to keep its efficiency. This is key to its speed, as hundreds of thousands of insertions are needed, constant balancing would greatly slow down the Trie generation.

Keys, as they are just letters, are unordered and are only dependent on the node above it, but independent from the leaves on the same level. Each node contains a boolean value, indicating whether the node is at the end of a word, and an unordered map, containing the letter as its key and a pointer to the next node. However, many implementations also use a standard map.

References

[1] Chavoosh Ghasemi, Hamed Yousefi, Kang G. Shin, and Beichuan Zhang. On the granularity of trie-based data structures for name lookups and updates. *IEEE/ACM Trans. Netw.*, 27(2):777–789, apr 2019.

[2] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, apr 2002.

[3] Julien Jorge. Effortless performance improvements in c++: std::unordered$_{m}ap$, 2023.

Map VS Unordered Map

Both types of maps are contained in the C++ standard library. A map is a "sorted associative container that contains key-value pairs with unique keys". They're implemented using Red-black trees, a self-balancing form of a binary tree.

An unordered map is an "associative container that contains key-value pairs with unique keys", implemented using a hash table. Hash-tables have a very fast search time, since once the hash is computated, the exact location is known, only when there are duplicated keys, are they keys placed into a bucket. That bucket is usually an array or vector that then needs to be searched in O(N).

Map Complexity

I chose to use an unordered-map instead of a standard map for two main reasons.

- It has a constant time search. Insertion order is not important in a trie, so this greatly increases search speed.
- There is no need for balancing. Hundreds of thousands of words need to be inserted and stored. Constant balancing when inserting each letter of each word would greatly decrease generation time when this isn't an issue with an unordered map.

	Map	Unordered_Map
Structure	Ordered by Keys	Unordered, associative
Implementation	Red-Black Tree	Hash Table
Balance	O(1) Average O(log N) Worst Case	Not Needed
Insertion	Log(n) + Rebalance	O(1) Average O(N) Worst Case
Search	Log(N)	O(1) Average O(N) Worst Case
Deletion	Log(N) + Rebalance	O(1) Average O(N) Worst Case
Space	O(N)	O(N)

Methods For Insertion

All words are stored in a text file of 370,105 words. My first solution to loading these into the Trie was to first read all the words into a vector and then iterate through that vector, inserting each word into the tree. However, the problem with this is that the vector has to update its size every time it's filled, which will happen thousands of times as the words are loading. This is an expensive process. 25% of the needed size is left empty when generating the new vector, taking up a large amount of memory. This would reduce the chances of the program running on low-end hardware, where memory space is more valuable.

An optimization to this is to use a linked list. After the vector's creation, it's only accessed in a linear fashion. The main benefit gained from using a vector, it's fast, indexed access, is never used.

A linked list would allow easy iteration from start to end, the only access needed to the structure. When a new word is added, it's just added as a new node, with its memory address being pointed to by the previous node. When loading the words into the Trie, one iteration is used throughout the list. This would increase loading speed as there's never a need to resize and allocate new memory to the list.

Final Insertion Optimisation

A better optimization is to avoid loading the words into anything but the Trie. When the file is opened, as each word is read, InsertWord(word) is called. This method consistently improves performance times as it reduces the need for any *middle-man*.

To measure the speed difference between the two methods, I created a timer class, using the Chrono package, to accurately measure the time taken to insert the entire file. 5-20 iterations of insertion were measured and the mean was calculated to give a comparable performance.

Mean Measured Performance Between Two Methods When Inserting All Words

Number of Iterations	Vector Method (ms)	File Read Method (ms)
5	2481.9	2097.5
10	2664.1	2133.7
15	2823.9	2116.3
20	2808.3	2230.6

Search Method Called on Each Word as its Inserted

```
bool Search(TrieNode* root, const char word[])
{
    TrieNode* currentNode = root;
    for (int i = 0; i < std::strlen(word); i++)
    {
        for (std::pair<char, TrieNode*> el : currentNode->children)
        {
            if (word[i] == el.first)
            {
                if (i == (std::strlen(word) - 1) && currentNode->children[el.first]->IsWord)
                {
                    return true;
                }
                currentNode = currentNode->children[el.first];
            }
        }
    }
    return false;
}
```

Figure 2. Insertion Algorithm

Diagnostic Report

Visual Studio 2022's performance analysis provides useful insights into the final performance of the program.

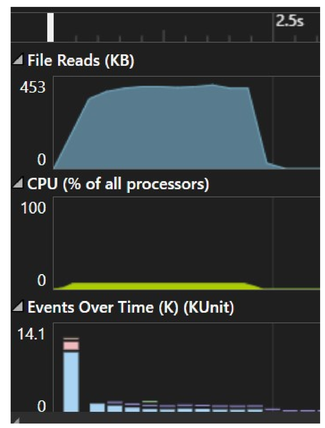

Figure 3. Diagnostic Report

Shown are the first 2.7 seconds of the program running. The stats measured are:

- Files Reads Data Read
- CPU% used
- Events

CPU usage peaks at 6% as the insertion of the entire words.txt file is being loaded into the Trie. The file reads peaks at 412KB.

After the insertion is over, any searches or further insertions cause no further spikes in CPU usage. This is the major benefit of an amortized complexity of O(1) for search and insertion time.