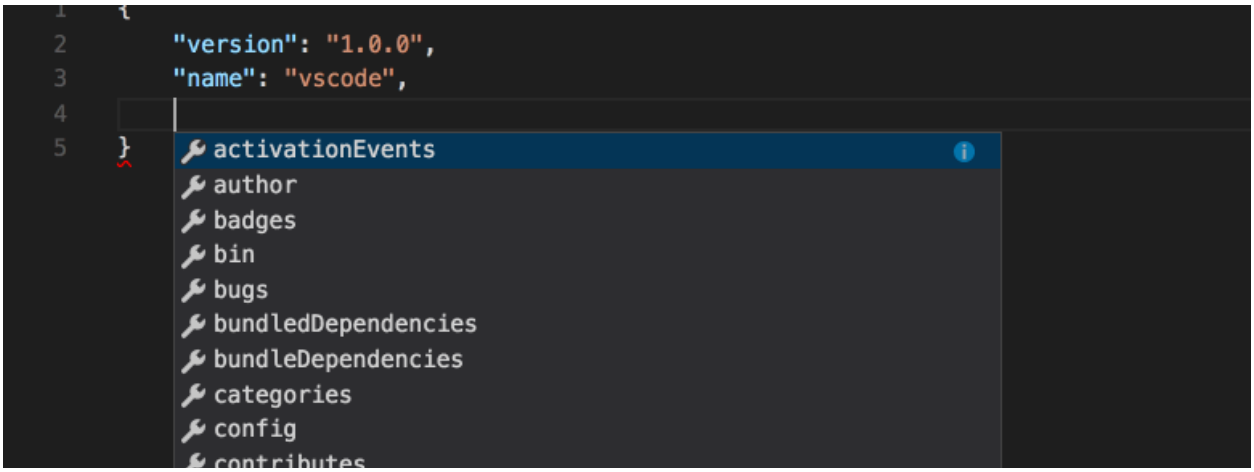


Autocomplete

Introduction

Autocomplete, or **word completion**, or say, **predictive text** is a feature in which an application predicts the rest of the word a user is typing.

Autocomplete speeds up *human-computer interaction* when it correctly predicts the word a user intends to enter after only a few characters have been typed into a text input field. It works best in domains with a limited number of possible words such as *command line interpreters*, when some words are much more common such as in *e-mails*, or writing structured and predictable text such as in *source code editors*.

Application

▼ Web Browsers

Autocomplete is a **default** feature of most modern *web browsers*. It anticipates what you are typing and suggests a word or phrase based on the activity of other users and your history. If you press enter , the application automatically completes your typing with the remainder of the suggested text.

- ▼ Email Software
- ▼ Search Engines
- ▼ Source Code Editors
- ▼ Word Processors
- ▼ Database Query Tools
- ▼ Command-Line Interpreters

Algorithms

An auto-complete feature can be implemented by searching a list of words and returning all words that begin with a given prefix. With the huge volumes of data being processed around the world in real-time, autocomplete is required to be accurate and fast using more efficient algorithms to **retrieve** and **rank** search phrases depending on the given input.

There are various approaches to implement an autocomplete feature using various algorithms, from the most naive ones to the others which are complex yet intriguing.

The most common algorithm used to implement autocomplete involves the use of **TRIE** data structure which is efficient in terms of both memory as well as time.

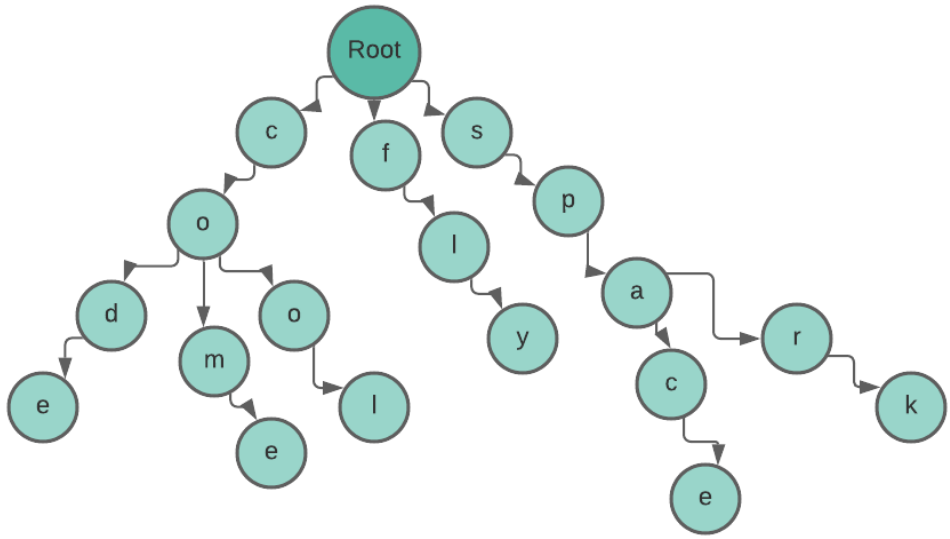
> 💡 Many autocomplete algorithms learn new words after the user has written them a few times, and can suggest alternatives based on the learned habits of the individual user.

TRIE

Problem - Given the prefix of user's search query, we need to give him all recommendations to autocomplete his query based on the strings stored in the TRIE.

Also known as the Prefix Tree or Radix Tree, a trie is basically a tree structure where each node is a character. Given below is an example trie, as one can see characters that are deriving from the same prefix, split into different branches. This trie contains the words - *code, come, cool, fly, space and spark*. The main plus point here is that no matter how many nodes there are in the trie, if you are looking for words that start with “co” then you will only need to search the branches that are formed from the parent nodes, “c” and “o”.

> The word count - number of words in the list matters, if the amount of words that are being searched are not significant, then there is little difference in actual time it takes to compute the possible words.



Here is great link to visualise how words are inserted and found in a TRIE.

Trie Visualization

https://www.cs.usfca.edu/~galles/visualization/Trie.html

▼ INSERT

As we insert keys in a TRIE, each input key is inserted as an individual node and perform the role of an index to further child nodes. If our input is a prefix of an existing string present in the TRIE, then we

can mark the new node as a leaf node to the pre-defined key. Otherwise, the new keys are constructed and the last node is marked as the leaf node.

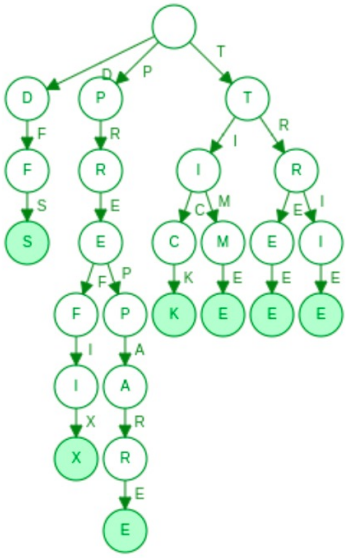
```
// Insertion of a node in a trie
void insert_string(string word)
{
    for(every character in word)
    {
       if(child node for character is NULL)
       {
           child_node = new Node();
       }
       current_node = child_node;
    }
}
```

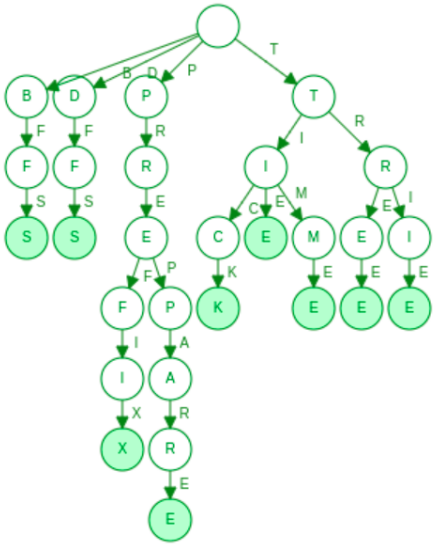
▼ SEARCH

A search operation starts from the root node and moves ahead by comparing characters. If any character does not exist, then no such string is found in the TRIE.

```
// Searching of a node
boolean search_string(string word)
{
    for(every character in word)
    {
        if(child node for character is NULL)
        {
            return false;
        }
    }
    return true;
}
```

Consider the trie given below, suppose you want to insert the word *tie* in the trie, we start from the root, make a recursive call to node 'T', passing the string "IE", next we make a recursive call to node 'I' and pass the string "E", next we find that there Child 'E' does not exist, create a new child node 'E' and make a recursive call to it passing the string "". We have reached the end of the string, and we set the current node to true *(leaf node)*.





▼ AUTOCOMPLETE

To implement the autocomplete feature, we extend the search function, once we have found the prefix in our trie, all we need to do is a depth first search or breadth first search from that node (as a head node) and keep a list of all complete strings encountered in the process. This list of strings is our list of autocomplete words.

```
def traversal(self, item):
        if self.leaf:
            print (item)
        for i in self.next:
            s = item + i
            self.next[i].traversal(s)

def autocomplete(self, item):
        i = 0
        s = ''
        while i < len(item):
            k = item[i]
            s += k
            if k in self.next:
                self = self.next[k]
            else:
                return 'NOT FOUND'
            i += 1
        self.traversal(s)
        return 'END'
```

The main advantage of using a trie data structure is that there is *no collision of different keys* in a trie. The best case time complexity is O(1) while the average and worst case is described as O(length of the prefix).

Here is the link to my implementation of the autocomplete feature using tries in *C++*

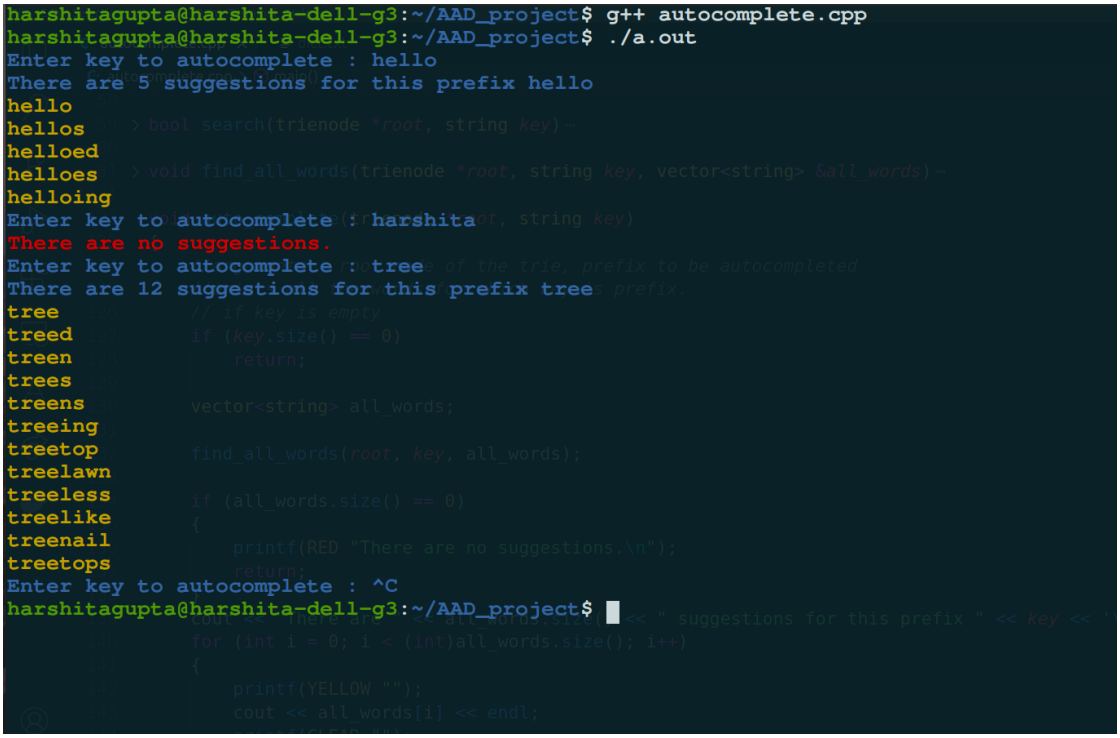
Algorithms-for-reals/Autocomplete at main · harshitagupta1512/Algorithms-for-reals

Contribute to harshitagupta1512/Algorithms-for-reals development by creating an account on GitHub.

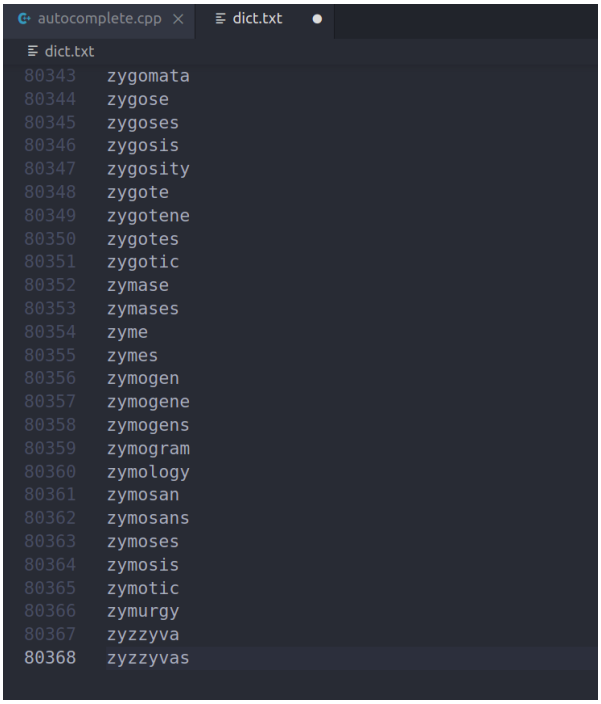
https://github.com/harshitagupta1512/Algorithms-for-reals/tree/main/Autocomplete

harshitagupta1512/
Algorithms-for-reals

1 Contributor 0 Issues 0 Stars 0 Forks



I used a dictionary with over 80k words, and every query gets processed in under a *millisecond*, thus trie data structure is one of the really efficient ways to implement autocomplete and autosuggestion features.



> *Linked list data structures be like - I know a guy who knows a guy.*