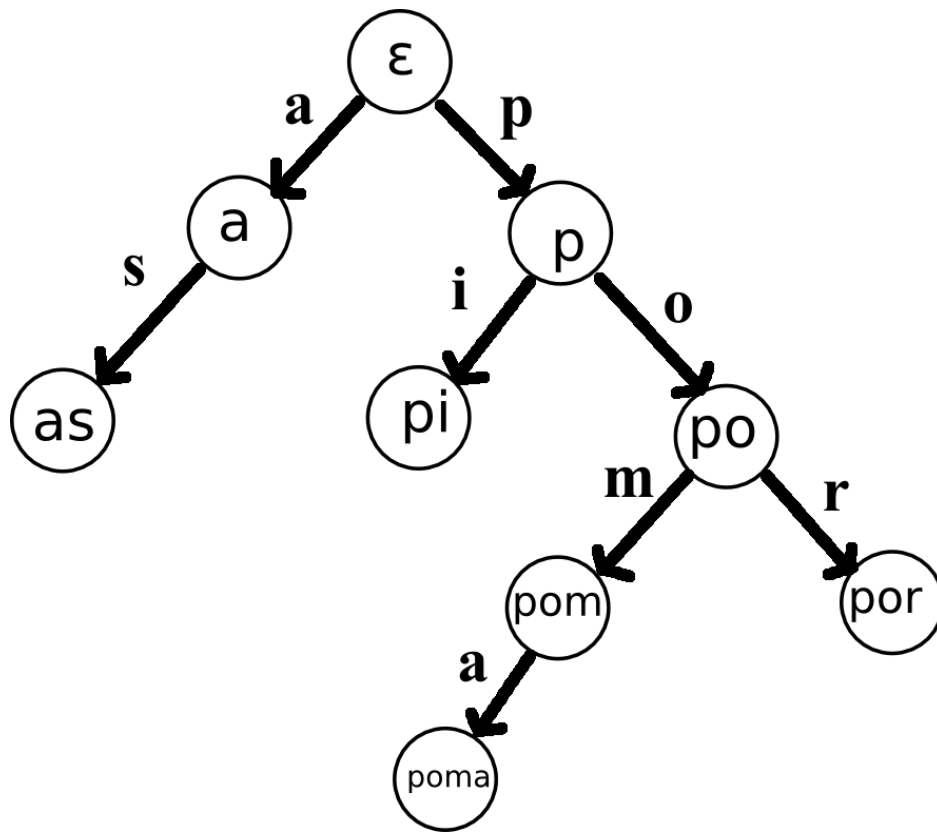


Trie Data Structure

Focus on storing and searching through a database/dictionary of words



David Ajuhan, Sahil Chadha, Cody Giroux, Skye Vrucak

Fall 2023
CSC - 212

Table of Contents

Introduction to Tries.....2

Basic Structure Overview

Our Project.....3

Functionality / Use

Methods.....4

Search and Insert

Implementation / Proofs.....6

Dictionary Datasets

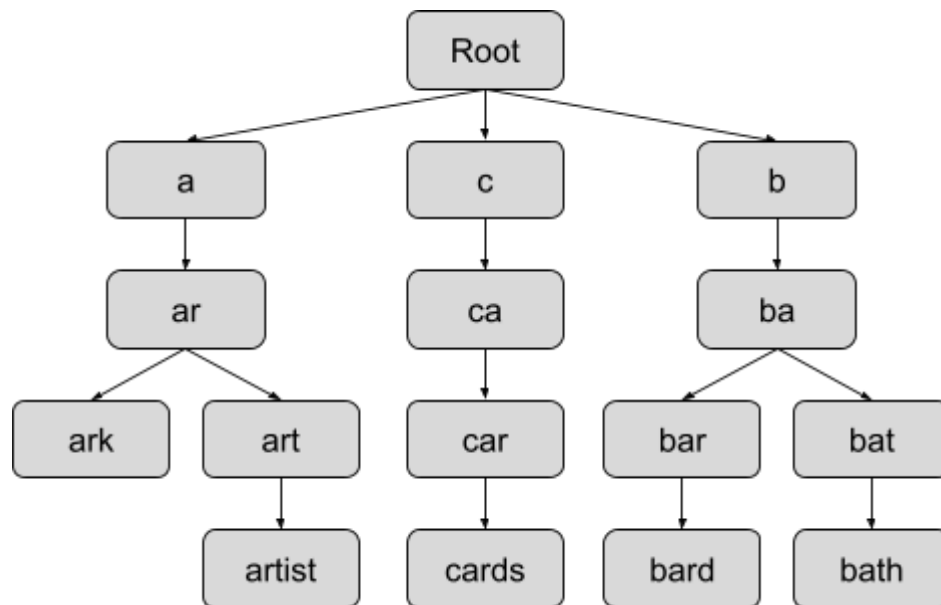
Contributions..... 9

Conclusion..... 10

References..... 11

Introduction to Tries

A trie (sometimes pronounced ‘try’ to avoid confusion), is a tree-like data structure that allows for the efficient locating of specific keys from within a set, and the re(trie)val of the contained data. Trie’s use a method of key-lookup where strings are used as keys, but instead of the nodes being linked together by the entire key– they are instead linked by the individual characters. These nodes do not store their associated key in the node like within a binary search tree, but instead use the position of the node to define the key it is associated with. This way, the value of each key is distributed throughout the entire structure, and also where not every node will actually contain a value. With this logic, all the children nodes of a parent share a prefix where characters match up to a certain degree until branching off within the structure. This prefix matching process is what we decided to incorporate in our project.



Our Project

Our task was to implement a real-world application using the trie data structure. We took interest in implementing both a database lookup and an autocomplete function using the structure. Autocomplete is used on many devices as it's a useful feature when entering in text. For some examples, in your IDE, you can type a couple of letters, hit 'TAB' and a command will be automatically pasted in for you. On your phone, signing up for an account wherever, and it asks for your email address? As you type in your name (assuming your name is at the start of your email) your email automatically shows up, thanks to your phone's personal dictionary. This is possible due to the prefix matching that trie's are commonly known for being efficient at.

The database implementation simply requires the user to upload their own text file that follows the proper formatting. The contents will become the trie and the user can then search for words within the tree. The program will output if they exist within the trie, how many times the word was repeated, and how many times it's used as a prefix for a word. The autocomplete implementation relies on a text file sourced from <https://github.com/dolph/dictionary/tree/master>. The 'popular.txt' file from that repository is turned into a trie acts like a dictionary, where the letters that the user inputted are used to find complete words using them as a prefix.

Methods

Class structure is similar to trees regarding the insertion of nodes and data. The main draw to trie's is that it is really efficient to search and retrieve data from the structure. The main operations that are to be looked into regard both insertion, and search.

Insert Analysis: Time complexity where n is equal to the number of characters in the string

```
void Trie::insert(const std::string& word) {
    TrieNode* current = root;
    //loops through all the characters in the word to check if the character in word exist as a key in the children map
    for (char ch : word) {
        ch = tolower(ch);
        if (current->children.find(ch) == current->children.end()) {
            current->children[ch] = new TrieNode();
        }
        current = current->children[ch];
        current->prefixCount++; // Increment prefix count at each node
    }
    current->isEndOfWord = true;
    current->count++;
}
```

Best Case: $O(1)$

Average Case: $O(n)$

Worst Case: $O(26*n) = O(n)$

To start, a node is created equal to the root in order to begin traversal through the structure. This is under the pretense that when the structure is created, a null root is automatically created as a blank string is required as the branching-off point for every other character in the alphabet. From the input word, the next character is iterated through and changed to lowercase. Once the next character is selected, it searches the map to see if the child already exists. If it doesn't, then a new node is created and added to the children map. The traversal node then iterates to the next character of the input word, and a counter increments showcasing how many times that word is used as a prefix for another word within the structure. For example, regarding the word "public," would return the number "7" as it is a prefix to the words "publically, publication, publications, publicist, publicity, publicly" and itself.

Search Analysis: Time complexity where n is equal to the number of characters in the string

```
std::pair<bool, int> Trie::search(const std::string& word) {
    TrieNode* current = root;
    int prefixCount = 0;
    //searches through the word and checks if the character in the word exist as a key in the children map and returns false
    // moves current to children node corresponding to character in word
    for (char ch : word) {
        ch = tolower(ch);
        if (current->children.find(ch) == current->children.end()) {
            return {false, 0};
        }
        current = current->children[ch];
        prefixCount = current->prefixCount;
    }
    return {current->isEndOfWord, prefixCount};
}
```

Best Case: $O(1)$

Average Case: $O(n)$

Worst Case: $O(26*n) = O(n)$

Once again a node is created equal to the root in order to begin traversal throughout the structure. The search function then iterates through each character of the inputted word and converts it to lowercase. If the character is not found, then the word does not exist in the structure's database. If the character is found then current will iterate to the next character and the process repeats itself. The function returns the count for how many times the word has been repeated.

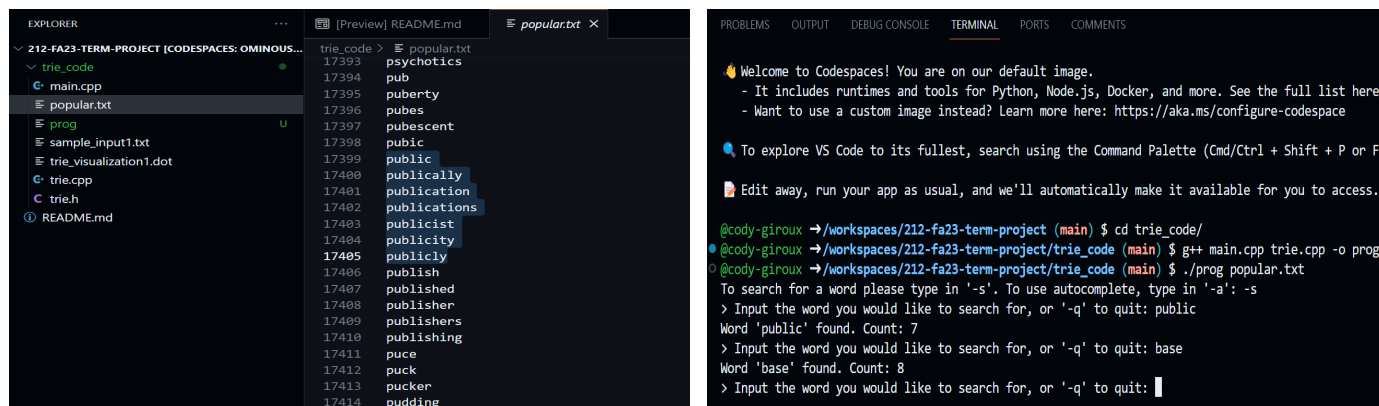
Implementation / Proofs

As mentioned earlier, the trie data structure can be used to implement a number of searching algorithms, from simple word searching to more complex functions like autofill and autocorrect. In our project, we decided to implement the autofill function, a function that will ask for part of a word, and output potential desired words that have the same prefix. The input word is taken as a command line argument, and the output is a queue of words from the trie that match the specified designators.

For all algorithms in our program, we first use a menu system, asking the user how they would like to proceed. Different inputs lead to different options, as seen below. Inputting “-s” will take the user to the next steps for the search algorithm, “-a” will use the autocomplete function, and “-q” will exit the program, leaving the user with a “Goodbye!” message in the command prompt. Next, the program will check if the input is valid by comparing the characters of the lines in the text file to letters, both capital and lowercase. If valid, we proceed to the actual function. Each function will loop asking for input and providing output until the user returns to the menu by typing ‘-q’.

The search function is fairly simple compared to other trie implementations. For our algorithm, we first prompt the user for the word we would like to search for. Next, we call the search function, using the inputted word as the parameter. This function will first set a pointer on the root of the trie, or the parent node. Then, it will iterate through the search word, and close in on the search word by heading down the trie node

by node. When reaching the leaf, the `isEndOfWord` indicator and the number of times that word is found is returned as a pair, and outputted by the main function. If a matching node is not found it will simply return false to signify the word does not exist within the trie.



The image shows a VS Code interface with two panels. The left panel displays a file explorer with a project named '212-fa23-term-project'. The file 'popular.txt' is open, showing a list of words with line numbers: 17393 psychotics, 17394 pub, 17395 puberty, 17396 pubes, 17397 pubescent, 17398 public, 17399 publically, 17400 publication, 17401 publications, 17402 publicist, 17403 publicity, 17404 publicly, 17405 publish, 17406 published, 17407 publisher, 17408 publishers, 17409 publishing, 17410 puce, 17411 puck, 17412 pucker, 17413 pudding. The right panel shows a terminal window with the following output:

```
Welcome to Codespaces! You are on our default image.
- It includes runtimes and tools for Python, Node.js, Docker, and more. See the full list here:
- Want to use a custom image instead? Learn more here: https://aka.ms/configure-codespace

To explore VS Code to its fullest, search using the Command Palette (Cmd/Ctrl + Shift + P or F1)

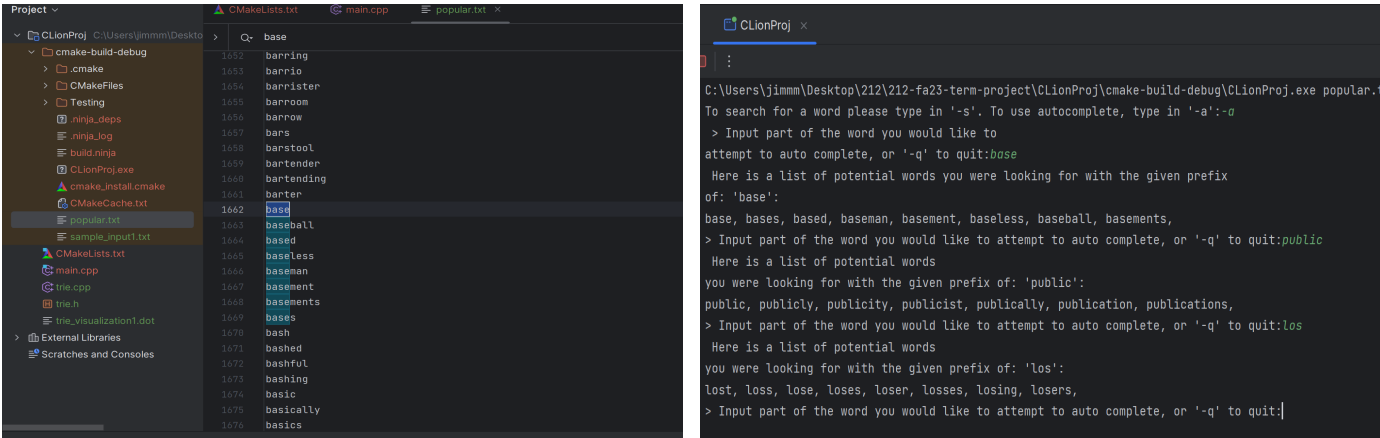
Edit away, run your app as usual, and we'll automatically make it available for you to access.

@code-giroux →/workspaces/212-fa23-term-project (main) $ cd trie_code/
@code-giroux →/workspaces/212-fa23-term-project/trie_code (main) $ g++ main.cpp trie.cpp -o prog
@code-giroux →/workspaces/212-fa23-term-project/trie_code (main) $ ./prog popular.txt
To search for a word please type in '-s'. To use autocomplete, type in '-a': -s
> Input the word you would like to search for, or '-q' to quit: public
Word 'public' found. Count: 7
> Input the word you would like to search for, or '-q' to quit: base
Word 'base' found. Count: 8
> Input the word you would like to search for, or '-q' to quit: 
```

Proof of Search function working on GitHub Codespace (VSCode online)

The autocomplete function was quite a bit more complicated to implement. It uses similar ideologies to the search function, as the algorithm will input a portion of the word and search for a node that contains the prefix that matches the search input. However, after searching for the node, the function then must return and output words matching the prefix. This search occurs as a Breadth First Search, where, using the prefix as the parent node, the function will search level by level and save all viable options as a queue of pairs, where the pair contains the node as well as the string of the word at the node.

Our implementation only allows 20 viable options to be outputted to the command prompt at a time, as more could cause runtime issues and lack organization in the output. These 20 words are prioritized as being those a minimal amount of letters off of the given prefix. After the algorithm ends, the main function saves the queue of words as a vector, and loops through the vector to output each word inside, leaving the user with 20 words to potentially use as autofill options.



Proof of Autocomplete function working on CLion

Contributions

Skye Vrucak (VioletVex on Github)	<ul style="list-style-type: none">● General Bugfixing and Error Checking● Implemented input/output checking and safety measures● Improved code readability● Found and edited dictionary data● Reworked main to be user-friendly● Wrote most of README.md
David Ajuhan (dajuhan on Github)	<ul style="list-style-type: none">● Created a spell checking function that provides a real world example of how a Trie structure is used● Implemented the portion of the main that calls that function● Helped give a skeleton of the autocomplete function
Sahil Chadha (schadha009 on Github)	<ul style="list-style-type: none">● Implementing the skeleton of the code, the search, insert, generateDotFile, and generateDot methods● Skeleton of the main.cpp and a few error fixes● Most of the header file
Cody Giroux (cody-giroux on Github)	<ul style="list-style-type: none">● Autocomplete function fulfilled● Allowing for choice of either search or autocomplete in main● Reworked header file and trie source file to allow for complete word tracking● Report formatting● Repository owner and kept general organization

Conclusion

Our group utilized a trie data structure that can parse through a dictionary of completed words. After all the words from the database were inserted into the structure, we were then able to create a text-autocomplete function using that data as a point of comparison. The words that are selected through this function, all share a common prefix with what has been input into the function. The reason why tries are so effective for this implementation, is due to the structure's tree-like nature, allowing for a fast run time. Inserting and searching through a trie only takes $O(n)$ runtime on average, and $O(1)$ at best, where 'n' is the length of input string. Without our structure, we would have to search through the dictionary in full to see what words share the same prefix as our input word. The trie allows us to traverse from the exact node where the last character of the input string branches outwards to other words in the dictionary.

This program is just one example of how tries can be used to quickly traverse through large amounts of text. Any dictionary, even from a small word bank can benefit from the trie's efficiency. Other implementations of this structure can be used in things like spell check. Such a task would regard searching if an input word that is not in the dictionary shares similarities to a certain threshold of other words in the provided dictionary. Another idea we had but not enough time to implement was a crossword puzzle solver. How this could have operated, would have us using a suffix matching method instead. Overall, tries are a great structure to use if the task at hand requires fast and efficient data storage and retrieval of strings and related sequences.

References

- <https://www.freecodecamp.org/news/how-to-validate-user-input-with-automated-trie-visualization/>
- https://petscan.wmflabs.org/?language=la&project=wiktionary&categories=Lingua%20Latina&ns%5B0%5D=1&sortby=title&interface_language=en&active_tab=tab_output&&doit=
- <https://www.codecademy.com/article/tree-traversal>