TRIE Tree: Autocomplete

Prepared for:

Professor J. Schrader

University of Rhode Island

Prepared By:

Matt Kyle, Student

Evan O'Neill, Student

Jacob Pierce, Student

Nitish Salvi, Student

CSC 212.0001: Data Structures & Abstractions

December 4th 2023

Table of Contents

Introduction

Preview:

This report will be organized in the following manner: First, there will be descriptions of a Trie, its use cases, and characteristics. Second, variations of the Trie will be described followed by a comparison of a Trie and another data structure. Third, is an implementation of a Trie as an autocomplete function.This section breaks down all files that were used to construct the autocomplete, interface, inputs, outputs, and a flow chart of the processes. Fourth, there will be discussion of the issues when using a Trie and some real world applications. Finally, there will be a conclusion, contributions, and references.

What are Trie Trees:

Trie Trees is a multiway tree data structure used for storing strings over an alphabet. Trie Trees are used to store large amounts of strings, and by using trie, they can efficiently pattern match. By using a common prefix, for example, in the words “all”, “apple”and “art”(a), the structure would have all those words coming from a common node. Trie trees are used as they are more efficient at storing strings than a binary tree. Using a common prefix allows for the structure to have a run time of o(N*L) (for transversal), where N is the number of words and L is the average length of the word.

Use Cases:

- Fast Search: A search for a key can be conducted by traversing down from the root, and search time is directly proportional to the length of the key. This makes Tries effective at searching large datasets.
- Space-efficient: Tires only store the characters that are present in the keys and not the entire key itself, making them ideal for storing large dictionaries and lexicons.
- Efficient insertion and deletion: Trie trees allow for simple deletion and insertion of nodes. keys can be quickly inserted or deleted.
- Efficient sorting: Tries support fast search and insertion operations, which make them ideal for sorting.
- Spell Checkers: If the word typed does not appear in the dictionary, then it shows suggestions based on what you typed. Tire stores the data dictionary and makes it easier to build an algorithm for searching the word from the dictionary and provides the list of valid words for the suggested prefix.

Key Characteristics of Tries:

The Trie, like other data structures, has certain characteristics that make it notable. Like binary search trees, a Trie is a tree data structure with nodes and edges. Unlike other tree structures, it can store multiple child nodes. Each node in a Trie represents a character while the edges display a relationship between nodes. These edges are labeled by the nodes that make them and form a path from the root to the node. As mentioned previously, the Trie is space-efficient. This is due to its dynamic nature. It grows and shrinks as the number of keys increases or decreases. Alongside these, the Trie is a flexible data structure as it can store numbers, custom objects and characters. Tries can also store binary digits.

Trie Variations:

The Trie data structure also contains multiple variations. Some common variations are Compressed Tries, Radix Trees, Suffix Tries, and Hash Tries. Compressed Tries aim to reduce space complexity by having common prefixes stored in nodes rather than individual nodes for individual characters. Radix Trees aim to improve space efficiency by merging nodes with only one child. It makes a Trie more complex but works especially well for IP routing. Suffix Tries are like Tries but store a string in reverse order to sort by the most common suffixes. Hash Tries improve the performance of the Trie by using a hash function to index. Each node stores a hash table compared to using fiexed-size arrays. These variations all improve upon the Trie in many different ways but all attempt to improve the efficiency of it.

Comparing the Trie Tree and Hash Table:

While the efficiency of the Trie tree structure has been discussed, a critical question about using them is how they work compared to other methods. One data structure that is often compared with a Trie is a Hash Table. Both are very similar in how they access and organize data however they are very different structures.

Advantages of Hash Tables:

- Easy to implement and understand
- Hahs provides better synchronization than other data structures

- Hash tables are more efficient than search trees or other data structures
- Hash provides constant time for searching, insertion, and deletion operations on average.
- The system will already have a well-optimized implementation faster than tries for most purposes.
- Keys need not have any special structure.
- More space-efficient than the obviously linked trie structure

Disadvantages of Hash Table over Trie:

- Hash is inefficient when there are many collisions.
- Hash collisions cannot be avoided for larger set of keys
- Hash does not allow null value.

Run Time Complexity:

- Time for Insertion: O(1)
- Time for Deletion: O(1)
- Time for Searching: O(1)

Advantages of Trie trees:

- Predictable O(n) lookup time where n is the size of the key
- Lookup can take less than n time if it’s not there
- Supports ordered traversal
- No need for a hash function
- Deletion is straightforward
- You can quickly look up prefixes of keys, enumerate all entries with a given prefix,etc

- Can efficiently do prefix search (or auto-complete) witn Trie.
- Can easily print all words in alphabetical order which is not easily possible with hashing.
- There is no overhead of Hash functions in a Trie data structure.
- Searching for a String even in the large collection of strings in a Trie data structure can be done in O(L) Time complexity, Where L is the number of words in the query string.

Disadvantages of Trie over Hash Table:

- The main disadvantage of the trie is that it takes a lot of memory to store all the strings. For each node, we have too many node pointers which are equal to the number of characters in the worst case.
- An efficiently constructed hash table has O(1) as lookup time which is way faster than O(L) in the case of trie where l is the length of the string.

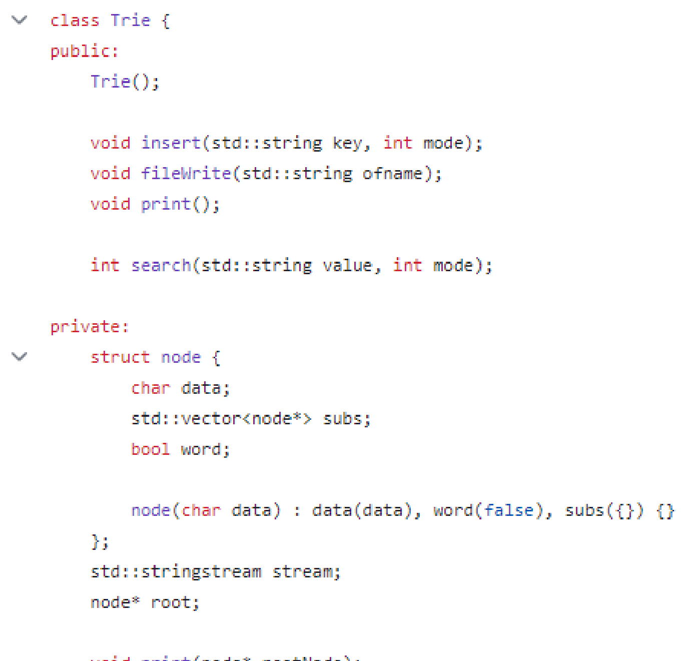
Run time complexity:

- Time for insertion: O(n)
- Time for Deletion: O(n)
- Time for Searching: O(n)

Implementation

Code:

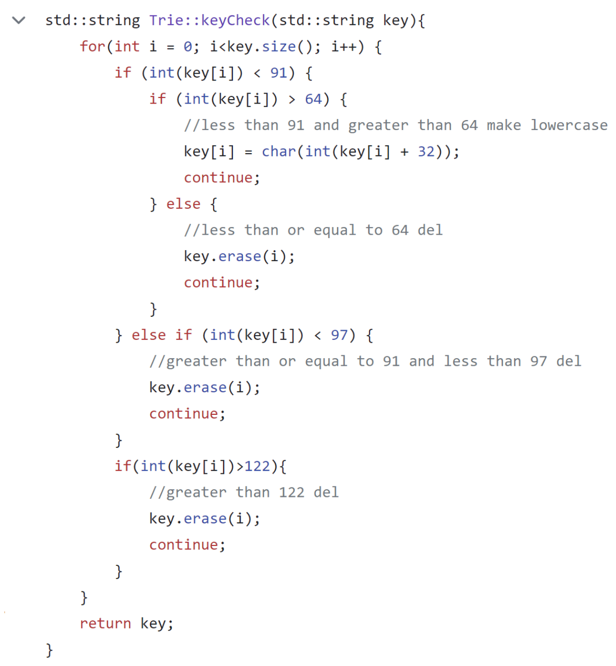
Trie.h: Within the private method, you can see the basic building blocks of the trie tree data structure. The structure is built using nodes that contain a char variable that is used to store the data or key of the node and



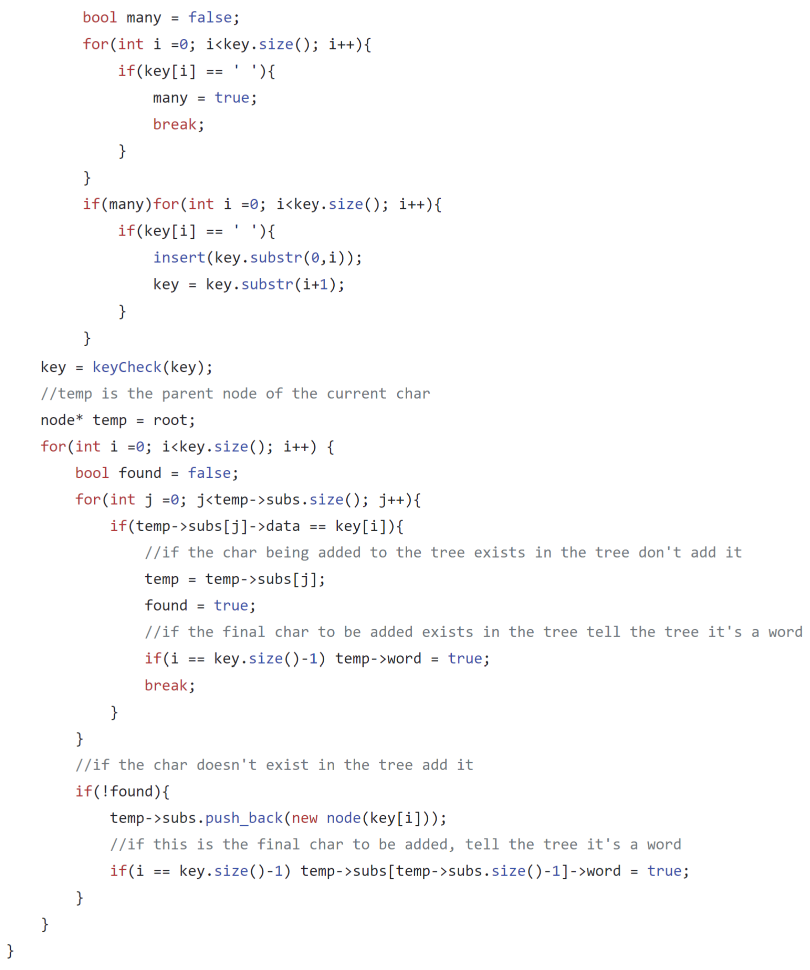
```
class Trie {
public:
    Trie();

    void insert(std::string key, int mode);
    void fileWrite(std::string ofname);
    void print();

    int search(std::string value, int mode);

private:
    struct node {
        char data;
        std::vector<node*> subs;
        bool word;

        node(char data) : data(data), word(false), subs({}) {}
    };
    std::stringstream stream;
    node* root;
```

a bool variable that is used to define whether a series of nodes has formed a word. A vector of node pointers is then used to store the nodes when building the tree.

Trie.cpp: Within the .cpp file, there are two main functions that control the operation of the program. That being the “keyCheck” function and the “Insert” function. The keyCheck function is used to check whether the key needs to be made lower case and if the key needs to be deleted. If a key is less than 91 and greater than 64, the key is converted to lowercase, and the program continues. The program then checks to see if the key is greater than or equal to 91, less than 97, or greater than 122. The key is then deleted. The “Insert” function simply inserts nodes into the tree. The function, however, conducts multiple checks first. The insert function first checks to see if multiple entries are being passed through the function; if so, the function is rerun with the individual entry. Once that is run, the key check function is run, and a temp node pointer is created, which points to the



```cpp
std::string Trie::keyCheck(std::string key){
    for(int i = 0; i<key.size(); i++) {
        if (int(key[i]) < 91) {
            if (int(key[i]) > 64) {
                //less than 91 and greater than 64 make lowercase
                key[i] = char(int(key[i] + 32));
                continue;
            } else {
                //less than or equal to 64 del
                key.erase(i);
                continue;
            }
        } else if (int(key[i]) < 97) {
            //greater than or equal to 91 and less than 97 del
            key.erase(i);
            continue;
        }
        if(int(key[i])>122){
            //greater than 122 del
            key.erase(i);
            continue;
        }
    }
    return key;
}
```

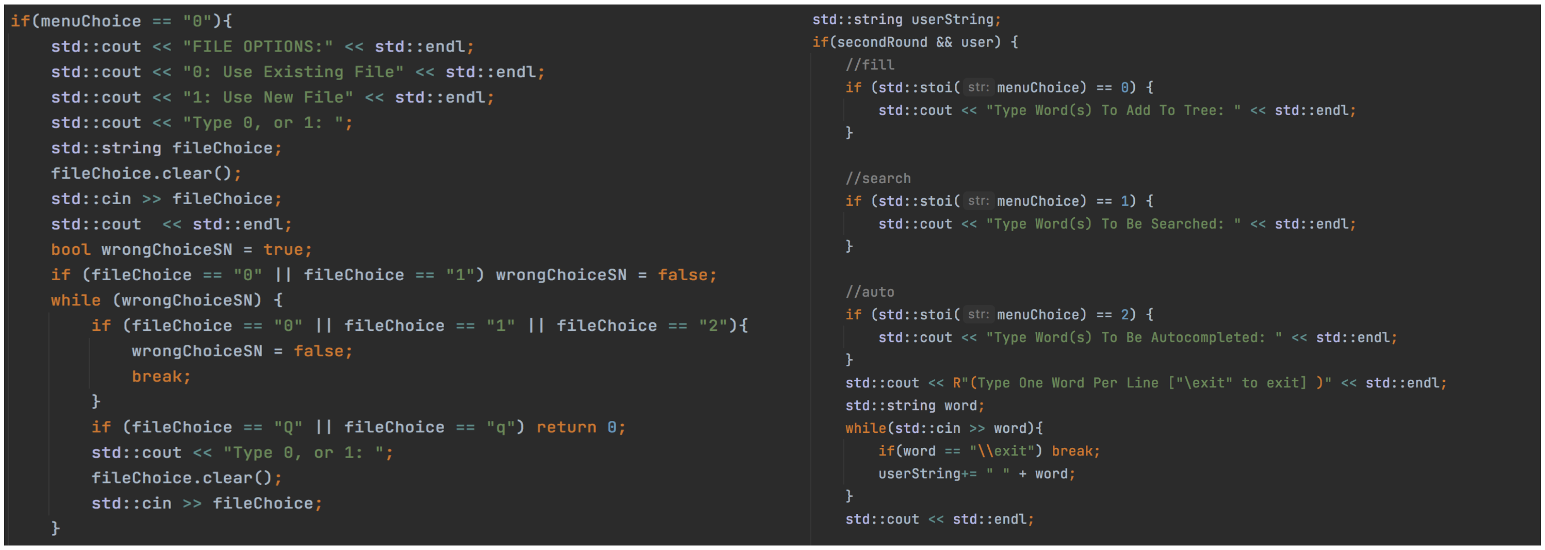


```cpp
        bool many = false;
        for(int i =0; i<key.size(); i++){
            if(key[i] == ' '){
                many = true;
                break;
            }
        }
        if(many)for(int i =0; i<key.size(); i++){
            if(key[i] == ' '){
                insert(key.substr(0,i));
                key = key.substr(i+1);
            }
        }
    key = keyCheck(key);
    //temp is the parent node of the current char
    node* temp = root;
    for(int i =0; i<key.size(); i++) {
        bool found = false;
        for(int j =0; j<temp->subs.size(); j++){
            if(temp->subs[j]->data == key[i]){
                //if the char being added to the tree exists in the tree don't add it
                temp = temp->subs[j];
                found = true;
                //if the final char to be added exists in the tree tell the tree it's a word
                if(i == key.size()-1) temp->word = true;
                break;
            }
        }
        //if the char doesn't exist in the tree add it
        if(!found){
            temp->subs.push_back(new node(key[i]));
            //if this is the final char to be added, tell the tree it's a word
            if(i == key.size()-1) temp->subs[temp->subs.size()-1]->word = true;
        }
    }
}
```

memory address of the root node. Using the public method of the insert function to transverse the structure recursively, it searches through the given key. If the key already exists within the tree, it is not added. If it is not found, the key is added to the appropriate root node, and if it is the final key to be added to the branch, a node is added to the end of that tree, and its bool variable is set to true, meaning we have a completed word.

Main.cpp:

The main file starts by taking a user-inputted CLA to represent the input file the user would like to perform on. The purpose of the main function is to display the menu of options the user can choose from to perform different operations. Nestled within these menu displays are the calls to the trie.cpp file, which contains all of the functionality for performing the user-specified operations. Each of the options have their own path and are sorted below their respective cout statements to improve readability. Below are two of the mere many examples of the menus we created to provide a hassle free experience for the user en route to using our autocompletion.

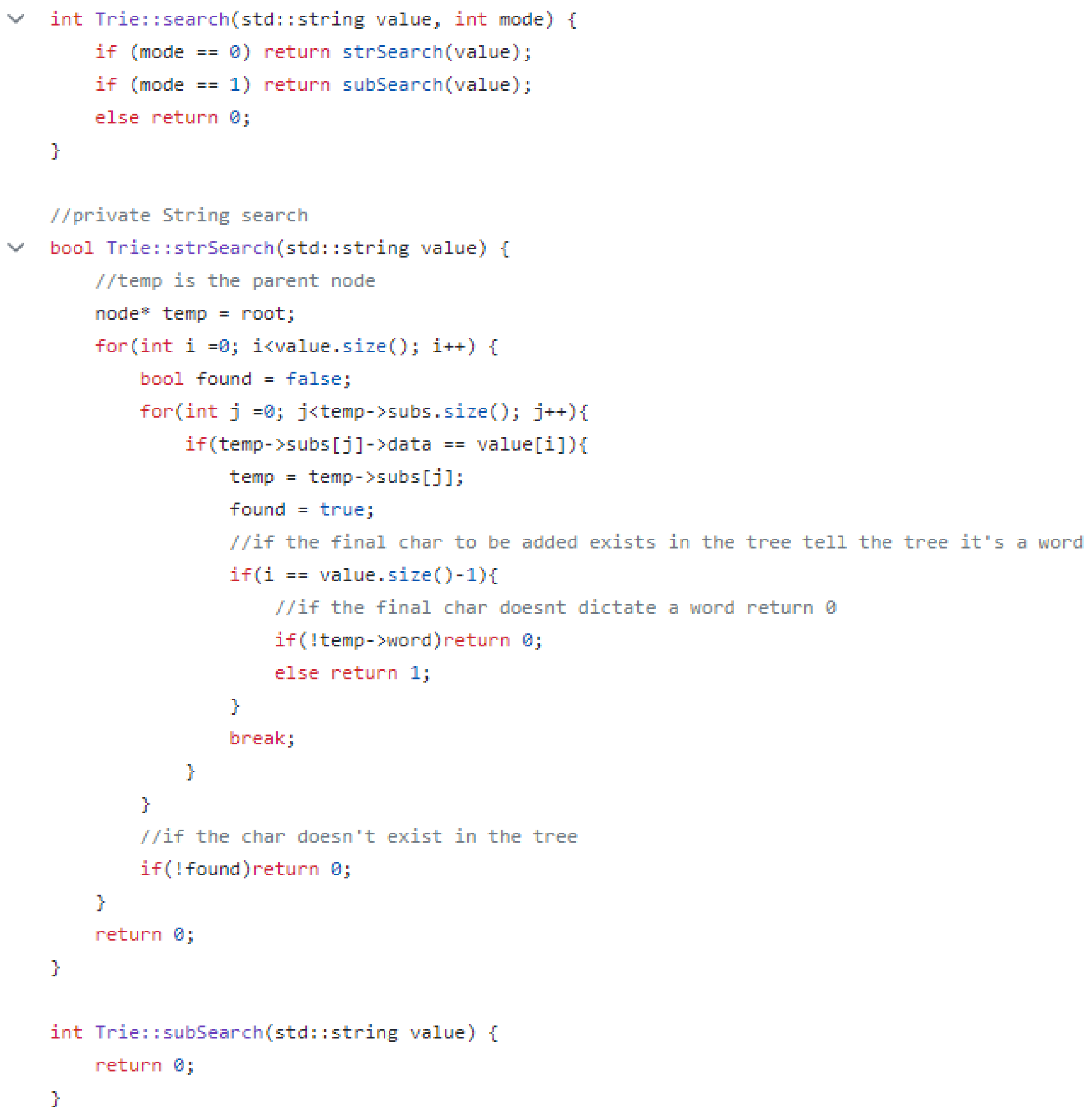


```
if(menuChoice == "0"){
    std::cout << "FILE OPTIONS:" << std::endl;
    std::cout << "0: Use Existing File" << std::endl;
    std::cout << "1: Use New File" << std::endl;
    std::cout << "Type 0, or 1: ";
    std::string fileChoice;
    fileChoice.clear();
    std::cin >> fileChoice;
    std::cout  << std::endl;
    bool wrongChoiceSN = true;
    if (fileChoice == "0" || fileChoice == "1") wrongChoiceSN = false;
    while (wrongChoiceSN) {
        if (fileChoice == "0" || fileChoice == "1" || fileChoice == "2"){
            wrongChoiceSN = false;
            break;
        }
        if (fileChoice == "Q" || fileChoice == "q") return 0;
        std::cout << "Type 0, or 1: ";
        fileChoice.clear();
        std::cin >> fileChoice;
    }
```

```
std::string userString;
if(secondRound && user) {
    //fill
    if (std::stoi( str: menuChoice) == 0) {
        std::cout << "Type Word(s) To Add To Tree: " << std::endl;
    }

    //search
    if (std::stoi( str: menuChoice) == 1) {
        std::cout << "Type Word(s) To Be Searched: " << std::endl;
    }

    //auto
    if (std::stoi( str: menuChoice) == 2) {
        std::cout << "Type Word(s) To Be Autocompleted: " << std::endl;
    }
    std::cout << R"(Type One Word Per Line ["\exit" to exit] )" << std::endl;
    std::string word;
    while(std::cin >> word){
        if(word == "\\exit") break;
        userString+= " " + word;
    }
    std::cout << std::endl;
```

Autocomplete:

Our program will use the Trie Tree data structure to autocomplete words based on the characters entered. The program will give a list of characters 1–10 with potential words based on the characters provided. By giving the program a dictionary of words, we create the Trie structure. By using the modified search program "subsearch" (Autocomplete), the function searches the tree based on the characters given. For example, if the characters given were "al"

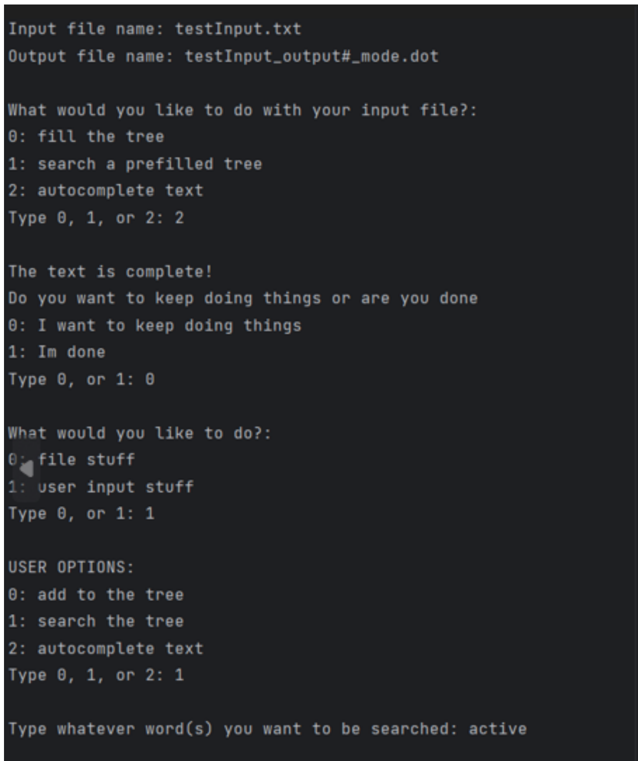


```
int Trie::search(std::string value, int mode) {
    if (mode == 0) return strSearch(value);
    if (mode == 1) return subSearch(value);
    else return 0;
}

//private String search
bool Trie::strSearch(std::string value) {
    //temp is the parent node
    node* temp = root;
    for(int i =0; i<value.size(); i++) {
        bool found = false;
        for(int j =0; j<temp->subs.size(); j++){
            if(temp->subs[j]->data == value[i]){
                temp = temp->subs[j];
                found = true;
                //if the final char to be added exists in the tree tell the tree it's a word
                if(i == value.size()-1){
                    //if the final char doesnt dictate a word return 0
                    if(!temp->word)return 0;
                    else return 1;
                }
                break;
            }
        }
        //if the char doesn't exist in the tree
        if(!found)return 0;
    }
    return 0;
}

int Trie::subSearch(std::string value) {
    return 0;
}
```

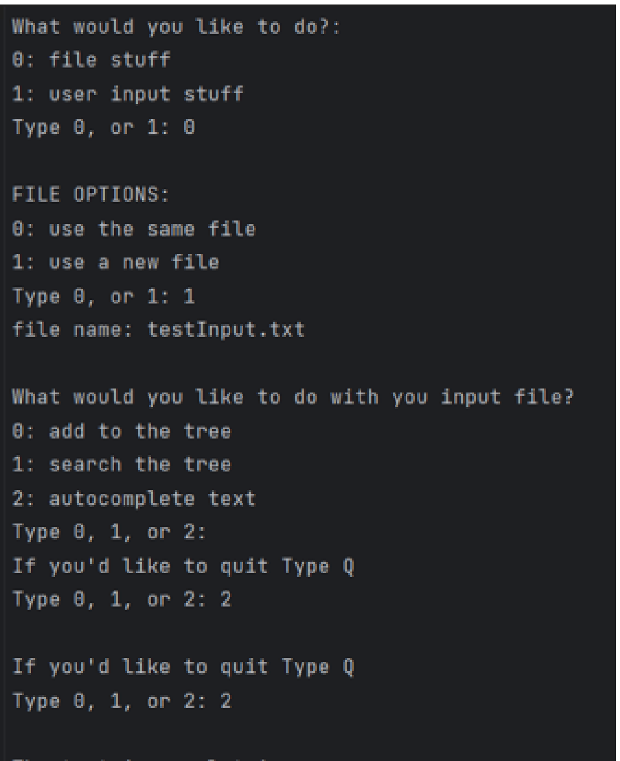
the program would look at the root for “a” and then follow that branch to find “l” once found (if not, a message will be returned to the console telling the user there were no words found), the function will look at all the branches stemming from that “al” root and will return a word from that root whenever a true bool value is found in a node.

Interface:

When a user runs the program, two options are presented: one prompts the user to enter a file, and the other brings them to the second interface, which presents the user options. When the user inputs a file, a .dot is created with the selected mode. The interface simply prompts the user with three options: to fill the tree with a given number of words, to search an already established tree structure for a given word, and lastly, to use the autocomplete feature (USER OPTIONS). If an option is selected, the program will run and output to the .dot file. A short message is then displayed about the option's completion. This system allows for the program to be continuously rerun until a new file is needed to be imputed. The program must be restarted in order for the user to create a new tree.



```
Input file name: testInput.txt
Output file name: testInput_output#_mode.dot

What would you like to do with your input file?:
0: fill the tree
1: search a prefilled tree
2: autocomplete text
Type 0, 1, or 2: 2

The text is complete!
Do you want to keep doing things or are you done
0: I want to keep doing things
1: Im done
Type 0, or 1: 0

What would you like to do?:
0: file stuff
1: user input stuff
Type 0, or 1: 1

USER OPTIONS:
0: add to the tree
1: search the tree
2: autocomplete text
Type 0, 1, or 2: 1

Type whatever word(s) you want to be searched: active
```

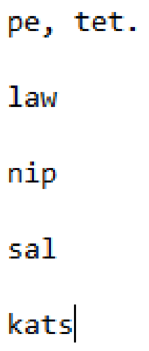


```
What would you like to do?:
0: file stuff
1: user input stuff
Type 0, or 1: 0

FILE OPTIONS:
0: use the same file
1: use a new file
Type 0, or 1: 1
file name: testInput.txt

What would you like to do with you input file?
0: add to the tree
1: search the tree
2: autocomplete text
Type 0, 1, or 2:
If you'd like to quit Type Q
Type 0, 1, or 2: 2

If you'd like to quit Type Q
Type 0, 1, or 2: 2
```

Input/Output:

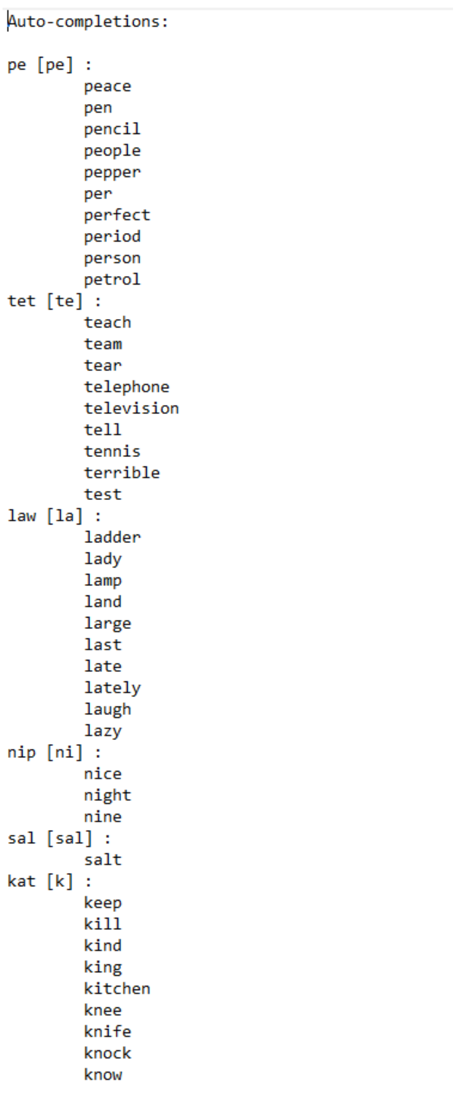
The program will search for the prefix given; if the given prefix does not exist within the tree, the program will search for the next best option. For example, in the below example, the output "kats" is not within our tree, nor is "kat" and so on. So the program searches for the longest prefix string; in this case, it is k.

Input autocomplete: Output autocomplete:



```
pe, tet.

law

nip

sal

kats
```

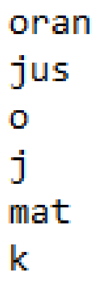
——-------------------------->



```
Auto-completions:

pe [pe] :
        peace
        pen
        pencil
        people
        pepper
        per
        perfect
        period
        person
        petrol
tet [te] :
        teach
        team
        tear
        telephone
        television
        tell
        tennis
        terrible
        test
law [la] :
        ladder
        lady
        lamp
        land
        large
        last
        late
        lately
        laugh
        lazy
nip [ni] :
        nice
        night
        nine
sal [sal] :
        salt
kat [k] :
        keep
        kill
        kind
        king
        kitchen
        knee
        knife
        knock
        know
```

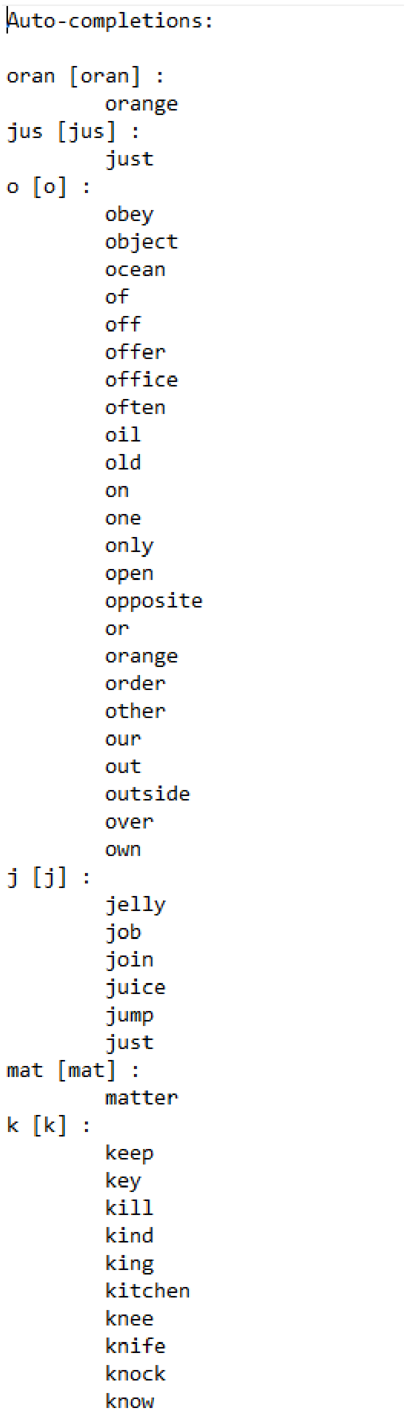
Input 2 autocomplete:

Output 2: autocomplete



```
oran
jus
o
j
mat
k
```

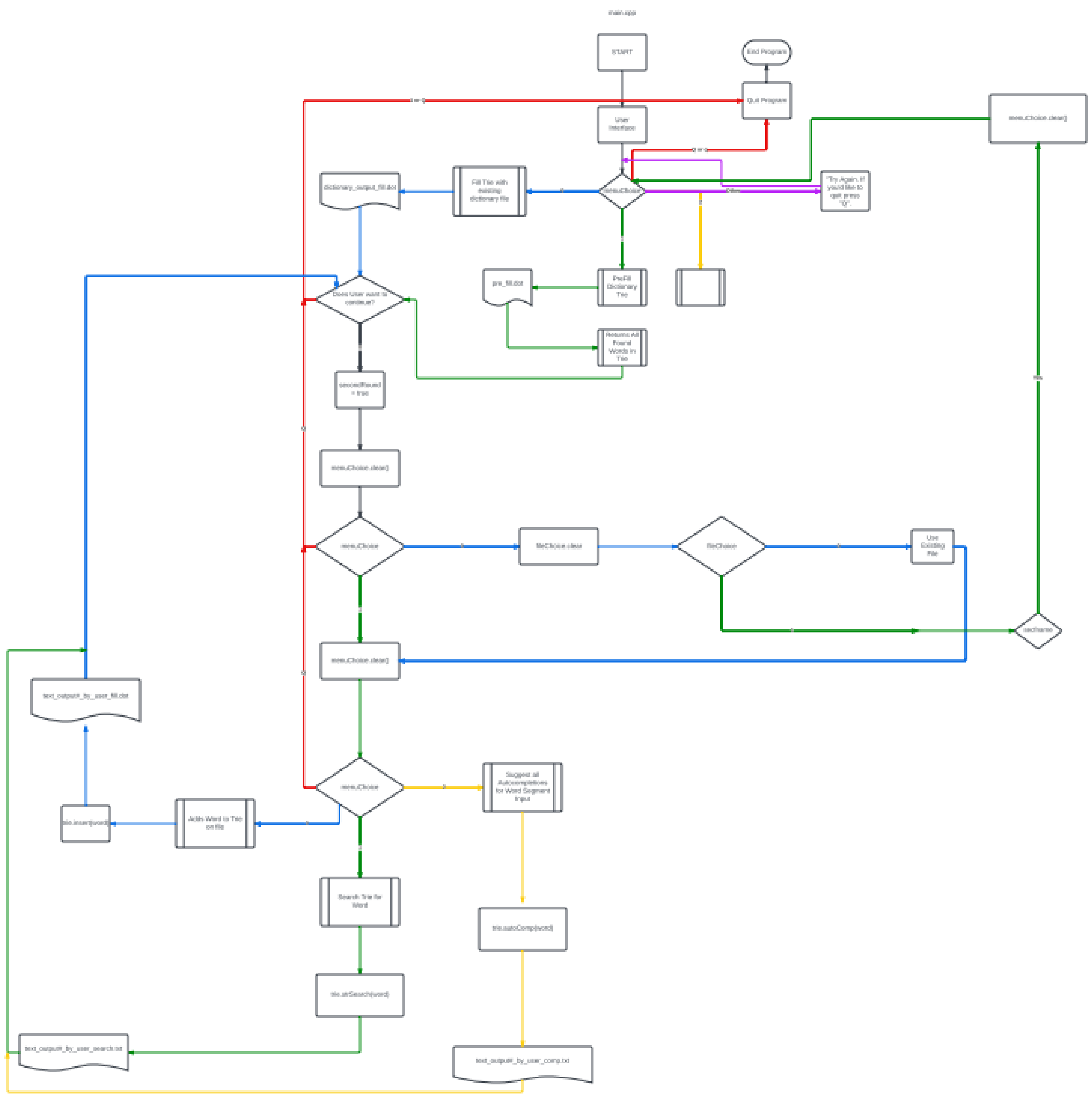
——------------------------------------>



```
Auto-completions:

oran [oran] :
        orange
jus [jus] :
        just
o [o] :
        obey
        object
        ocean
        of
        off
        offer
        office
        often
        oil
        old
        on
        one
        only
        open
        opposite
        or
        orange
        order
        other
        our
        out
        outside
        over
        own
j [j] :
        jelly
        job
        join
        juice
        jump
        just
mat [mat] :
        matter
k [k] :
        keep
        key
        kill
        kind
        king
        kitchen
        knee
        knife
        knock
        know
```

Flow Chart:

The flowchart below visualizes all the potential input/output scenarios of our program.


main.cpp
START
End Program
Quit Program
User Interface
menuChoice.clear()
menuChoice
"Try Again, if you'd like to quit press "Q"."
Other
dictionary_output_fill.dat
Fill Trie with existing dictionary file
Does User want to continue?
pre_fill.dat
PreFill Dictionary Trie
Returns All Found Words in Trie
secondRound = true
menuChoice.clear()
menuChoice
fileChoice.clear
fileChoice
Use Existing File
newFile
menuChoice.clear()
text_output_by_user_fill.dat
menuChoice
Suggest all Autocompletions for Word Segment Input
trie.insert(word)
Adds Word to Trie on file
Search Trie for Word
trie.autoComp(word)
trie.strSearch(word)
text_output_by_user_search.txt
text_output_by_user_comp.txt

Conclusion

Issues:

From conducting this project, we can surmise that when implementing the Trie tree data structure, some issues may arise, in particular when trying to traverse the tree. That was an issue that had to be solved before the implementation of the auto-complete could be concluded. Traversing the tree made it difficult to print the .dot file as well as search for words in the tree. These two issues were by far the most time-consuming and took up a good chunk of development time when it came to writing the program. Another issue came when adding a word to the structure. For example, when adding the word “keep” to the tree, the pointers should look like "k->e > e->p" instead of the root node being pointed to each of the keys in the word like “k->e, k->e, k->p”. This meant the program could not find words because, when a search was activated, the program only saw “ke”, “ke”, “Kp”. which caused the program to give incorrect outputs. However, with some research and debugging, these issues were resolved.

Real-World Applications:

Since Tries uses a prefix search method, there are plenty of scenarios where a Trie can be used. One of these scenarios is autocomplete. When a browser fills in a word as the prompt is given in the search bar, it may come up with results in the search bar before the prompt is finished. This is an example of autocomplete that utilizes a Trie. Another example of this is physically indexing through a dictionary. As a person searches for a word in the dictionary, they involuntarily recruit a Trie. By looking for the word “therefore”, finding the “T” section should be the first step. Then finding the “th” and so on until the word is found. One more real-world application is using a contacts app. By typing in a phone number, a list of contacts shows up based on the prefix numbers provided.

Finale Conclusion:

The Trie tree data structure is a useful data structure to have at your disposal. Given the structure's unique design to use characters as keys or data, it allows for greater manipulation of strings, which makes it a useful program for the manipulation and storage of string data. Furthermore, the range of applications for fast-search, auto-complete, space-efficiency, etc.The versatility of the Trie structure make it a staple data structure in the industry. The Trie tree structure allows for a greater manipulation of string data that allows for many useful functions that we see day to day to exist. Without the Trie tree structure many of the programs that are relied on day to day would not be as efficient and much more computationally demanding. Some may be completely inoperable. Making the Trie tree data structure an essential part of computer science and a necessary data structure to understand and be able to implement.

Contributions

Matt Kyle	Code, Debugging, and Repository
Evan O’Neill	Code, Debugging, and Repository
Jacob Pierce	Report, Presentation, and Repository
Nitish Salvi	Report, Presentation, and Repository

References

Online Graph Visualizer
https://dreampuf.github.io/GraphvizOnline/#digraph%20G%20%7B%0A%0A%0A%7D

LucidChart: Flowchart Creator
https://lucid.app/lucidchart/dcd03a0f-d83c-4ca3-bf8a-da2cc1256667/edit?page=0_0&invitationId=inv_cbbcba8b-0b9d-4bb0-9cc7-1af2ccd227ad#

Venki. (N/A). Trie | (Insert and Search). GeeksforGeeks.
https://www.geeksforgeeks.org/trie-insert-and-search/