Data Structures
and
Object Oriented Programming

Final Project Report

Compiled by:
L2CC
Nicholas Bryan (**2802523042**)
Kenny Krixiadi (**2802529191**)
Jerico Tjan (**2802521112**)

BINUS University International
Jakarta
2025

Table of Contents

Chapter 1
Project Specification

1.1. Project Description

For our final project, we concluded that with the ever-increasing rate of technological development and the rise in demand for greater efficiency, the tools needed to develop emerging technologies are also expected to be highly efficient. One of the tools in great demand is autocomplete for typed words, a feature often used not only in texting but also in search engines and various email programs, such as Outlook and Gmail.

With the sole purpose of developing an autocomplete system that is fully optimized for performance, both in terms of processing speed and memory, certain key factors play crucial roles in determining the performance of an autocomplete system. In this project, we aim to investigate how memory affects the performance of an autocomplete system, particularly when utilizing different data structures. Two types of data structures have been chosen for this experiment: the Trie and the Ternary Search Tree (TST).

1.2. Project Link

https://github.com/KennyKd/oogl

Chapter 2
Solution

2.1. Solution Design

Our proposed solution to this problem involves conducting a side-by-side comparison of both data structures using a large dataset of words to be autocompleted. After that, we will evaluate their accuracy, speed, and memory usage to determine the effectiveness of each data structure. The primary data structures used in this project are Tries and Ternary Search Trees, as they form the core of this experiment. To build it from an application's perspective, we decided to use the *javax.swing* package as our UI builder.

2.2. Data Structures

A. Trie

A trie is a data structure that is also known as a digital tree or prefix tree, usually used for fast retrieval on large data sets. The trie is an ordered tree that represents a finite alphabet set, usually the Latin alphabet consisting of 26 characters. In the way that it is built, words that have common prefixes share the same prefix data, which enables the tree to store many sequences efficiently. A node in a trie has 3 attributes, which are:

1. A character to store the letter,
2. An array of nodes to represent the children of the node (maximum length depends on the size of the alphabet set),
3. A boolean to determine whether the node is the end of a word or not.

B. Ternary Search Tree

A ternary search tree (TST) is a modified trie where the child nodes of a standard trie are ordered as a binary search tree. Unlike how a trie works, there are only 3 child pointers in TST: left, equal, and right. The ternary search tree also requires an ordered alphabet set, meaning a certain element comes before and/or after another element. By using that property, words are loaded into the tree by

their character's weight. If a word contains a letter whose weight is lower than the current pointer's letter, then it assigns itself to the current node's left pointer. On the other hand, if the letter's weight is bigger than the current pointer's letter, it assigns itself to the right pointer. If it still belongs to the same word, or it has the same prefix, it traverses through the middle pointer.

C. Performance Comparison

Below are the average-case complexities based on a theoretical perspective taken from the code of the trie and TST. Note that insertion, deletion, and search operation space complexities represent auxiliary space.

Feature		Trie	TST
Insertion	Time	O(n)	O(n * log a) n: length of word. a: alphabet size.
	Space	O(1), iterative	O(n), recursive
Search	Time	O(n)	O(n * log a)
	Space	O(1), iterative	O(n), recursive
Memory usage		O(N * K), hashmap N: number of words loaded. K: average length of words.	O(N * K) N: number of words loaded. K: average length of words.

Table 1. Trie and TST performance comparison

D. Data Structure Modification

To accommodate a large dataset efficiently in this project, both trie and TST implementations are modified to hold data on word relevance in the form of a frequency value. This value acts as information on the importance of a word, which is based on how often the word is searched. During the autocomplete

retrieval, the system will return all possible completions. However, rather than returning all of it, it will just prioritize the top K elements, which is defaulted to 5 and can be changed. This is done in order to enhance user experience by assuming the most likely search. To achieve this, the filtering is done by using a reversed priority queue, where each entry is a pair of the word and its frequency. This strategy ensures the system to be high performance despite large word datasets.

2.3. OOP

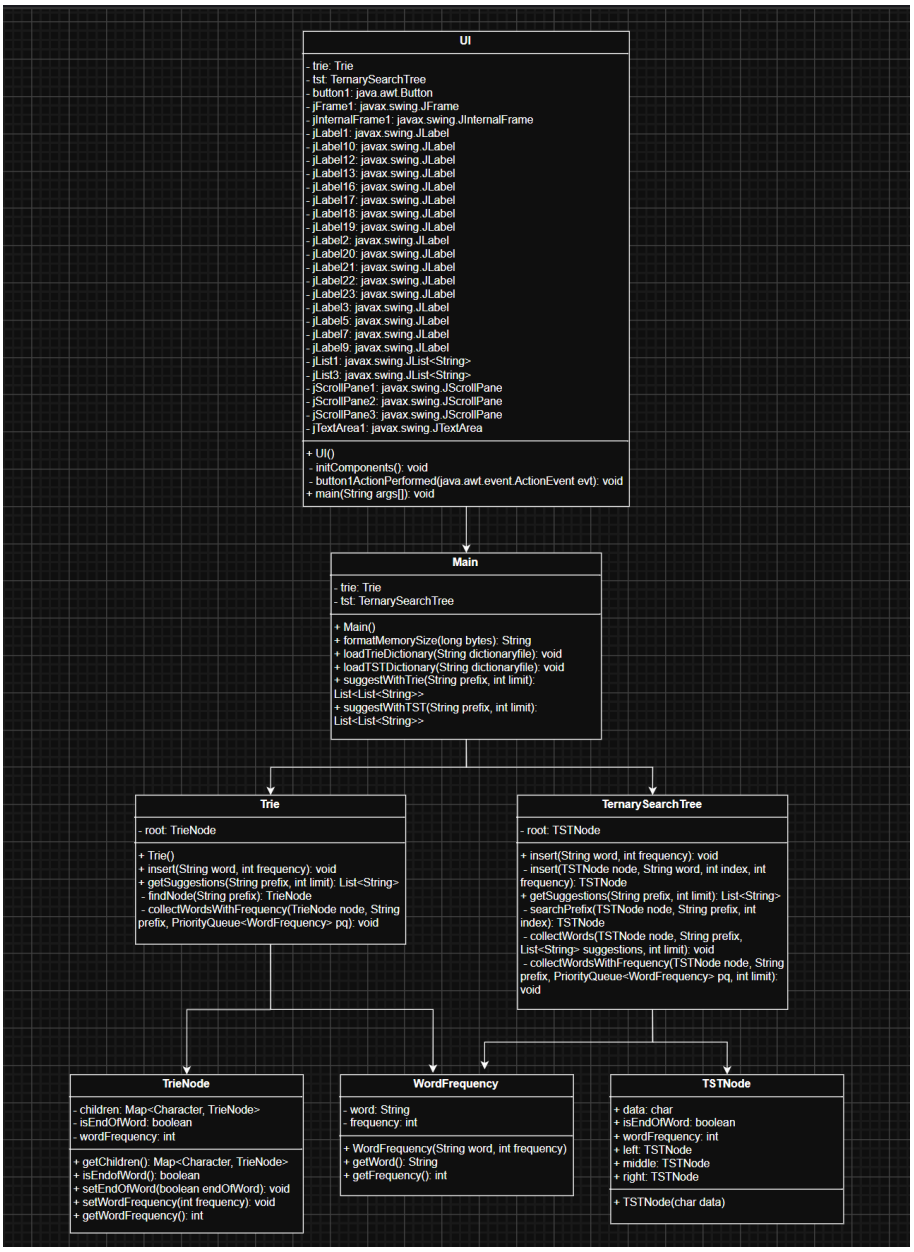


Figure 3. Class diagram

During the development of this program, we have integrated various styles from the Object-Oriented Programming (OOP) class. One of the main pillars of OOP, which is encapsulation, is used throughout the structure of classes and their data, protecting the class's data from being directly accessible by other classes [7]. Abstraction is also heavily used in the code's structure, which is evident in the UI class, showing only the necessary information provided by other classes, such as Main [8]. Different design patterns are also put into consideration, with one of them being the facade pattern, which is to hide the working logic from users and only show a simplified interface to use the working logic [9]. Another design pattern used in the code includes the observer pattern for the button in the UI class, which is used to receive input and notify other methods to run the hidden working logic [10]. All of the styles and design patterns used throughout the entire code aid both users in using the program and developers in understanding how the code works with more ease.

2.4. Runtime Results

The results are gathered by simulating 100 queries, each using different prefixes that do not make a complete word, ranging from 1 to 3 letters long. The dataset used for both the application and test runs consists of 6744 English words that are ordered by their frequency count, scaled down by 10000 units to consider the integer limit in Java. Measurements that are taken into account are initial memory usage, time and space taken to load the trie and TST, total time for 100 queries for both the trie and TST, average time per query, and total accuracy score. All time- and space-based measurements are done using the *java.lang.Runtime* class, while accuracy is calculated differently. Accuracy is based on the order of the given suggestions, which must align with the number of frequencies. We will deduct points from the data structure if any of the words' order is incorrect. In the end, accuracy will be the ratio of points to the total words suggested for the 100 queries.

Measurement		Trie	TST	Margin
Initial memory		2.18 MB		
Load	Time	126893500 ns	115248334 ns	11645166 ns
	Space	3.64 MB	584.22 KB	3.07 MB
Query	Total time	81164914 ns	17298210 ns	63866704 ns
	Accuracy	100.00%	90.89%	9.11%

Table 2. Trie and TST runtime results

2.5. Discussion

A. Theory and Practice

Based on the observed findings, there is a misalignment between the theoretical complexities and the runtime results. 3 metrics reflect this, which are discussed below.

1. Insertion time: The TST performs better despite its O(n * log a) complexity. The trie is slower, with a time of 126.8 ms, compared to the TST's 115.2 ms.
2. Search time: The TST performs better with 17.3 ms, while the trie is 4.7x slower.
3. Memory usage: The TST has better space allocation, with the entire tree occupying just 584 KB, while the trie occupies 3.64 MB.

The key reasons for these theory-practice differences are the usage of hashmaps in the trie's implementation. Standard-chain hashtable, which is Java's approach to hashmaps, is not optimized for cache performance, resulting in poor spatial locality [6]. This means the usage of a hashmap may introduce hashing overhead and cache misses due to the nodes being scattered in memory. To add, the effect of cache inefficiencies is detrimental to the overall performance of a system because each cache miss causes a delay of hundreds of clock cycles [6]. On the other hand, TSTs operate by direct pointer chasing, which works linearly and hence utilizes a single cache line to retrieve data [6]. Ultimately, this leads TSTs to have less latency. In terms of space, TST also performs better for a similar reason.

Using a hashmap means each of the trie's nodes needs to store hash values, buckets, and linked-list pointers, causing a memory bloat.

B. Accuracy

There are 2 reasons for the lower accuracy on the side of TST, both are caused by the filtering and the retrieval code. Because of this, there is no inherent reason coming from the data structure itself.

```
public List<String> getSuggestions(String prefix, int limit) {
   List<String> suggestions = new ArrayList<>();

      if (prefix.isEmpty()) { // Error handling for empty
prefixes.
       collectWords(root, "", suggestions, limit);
       return suggestions;
   }

   TSTNode lastNode = searchPrefix(root, prefix, 0);

   if (lastNode == null) {
       return suggestions;
   }

   if (lastNode.isEndOfWord) {
       suggestions.add(prefix);
   }

   collectWords(lastNode.middle, prefix, suggestions, limit);

   return suggestions;
}
```

Figure 2. *getSuggestions* function

From *Figure 2*, this function immediately adds any prefix that already makes up an entire word represented by the block of if-case *lastNode.isEndofWord*. This bypasses the filtering that happens at the end using the priority queue. Hence, all of the testing done on TST returns a score of 4 or 5 on each of its queries.

A much more subtle bug is seen in the results of the example query presented below.

1. Trie

   Prefix 'ob': 5 suggestions, 15500 ns, accuracy: 5/5

   Suggestions: [object, obtain, objective, obviously, obvious]

   Frequencies: [object:6982, obtain:3067, objective:2247, obviously:1570, obvious:1553]

2. TST

   Prefix 'ob': 5 suggestions, 9333 ns, accuracy: 5/5

   Suggestions: [object, objective, obligation, observation, obesity]

   Frequencies: [object:6982, objective:2247, obligation:1412, observation:1222, obesity:564]

Both data structures successfully score 5 points for the prefix “ob”. However, the TST is missing a higher frequency word, which is “obtain” with the frequency value of 3067. The cause is the code snippet below.

```
private void collectWordsWithFrequency(TSTNode node, String
prefix, PriorityQueue<WordFrequency> pq, int limit) {
    if (node == null || pq.size() >= limit) {
        return;
    }

    collectWordsWithFrequency(node.left, prefix, pq, limit);

    String word = prefix + node.data;
    if (node.isEndOfWord) {
        pq.offer(new WordFrequency(word, node.wordFrequency));
    }

    collectWordsWithFrequency(node.middle, word, pq, limit);
    collectWordsWithFrequency(node.right, prefix, pq, limit);
}
```

Figure 3. *collectWordsWithFrequency* function

Firstly, the retrieval is done by DFS, which is performed from left to right. This means that the words added to the priority queue in advance are those on the left

branch of the prefix node. This wouldn't be a problem if it weren't for the recursion's base case: *pq.size() >= limit*. This base case limits the words added to the priority queue by the limit, which is defaulted to 5. When it reaches 5, the program terminates before it has a chance to traverse all branches of the subtree, causing a possible miss in a higher frequency word.

2.6. Possible Improvements

1. Code Fixing

The code performed by the TST does work, but there are micro-bugs in the retrieval and filtering of the words, causing the TST to underperform in terms of accuracy.

2. Trie Enhancements

The trie data structure used in the autocomplete system could be enhanced in multiple ways, such as employing a radix tree instead of a standard trie. Another is to opt for a compact chain hashtable or a contiguous array instead of a separate-chained hashtable for better handling of cache, proven in [6] by offering significant space savings and reduction in space overhead.

2.7. Conclusion

In conclusion, the current project presents an extensive comparison of the Trie and Ternary Search Tree (TST) data structures in the context of an autocomplete system. The paper compared these structures on the following main performance aspects: speed, memory efficiency, and accuracy. Although the trie has better time complexity properties in theory, especially in search and insertion operations, practical measurements revealed that TSTs were much faster in terms of runtime and less demanding in terms of memory consumption due to their pointer-based representation and superior cache locality. The performance characteristics of the trie that relied on a Java hashmap created significant memory overhead and slowness because of the frequent cache misses.

The trie was always accurate in the suggestions, while the TST was found wanting in certain instances, owing to logical deficiencies in the filtering as well as traversal strategies. These bugs turned out not to be related to the underlying

structure, but to problems in the implementation of the suggestion-retrieval algorithm.

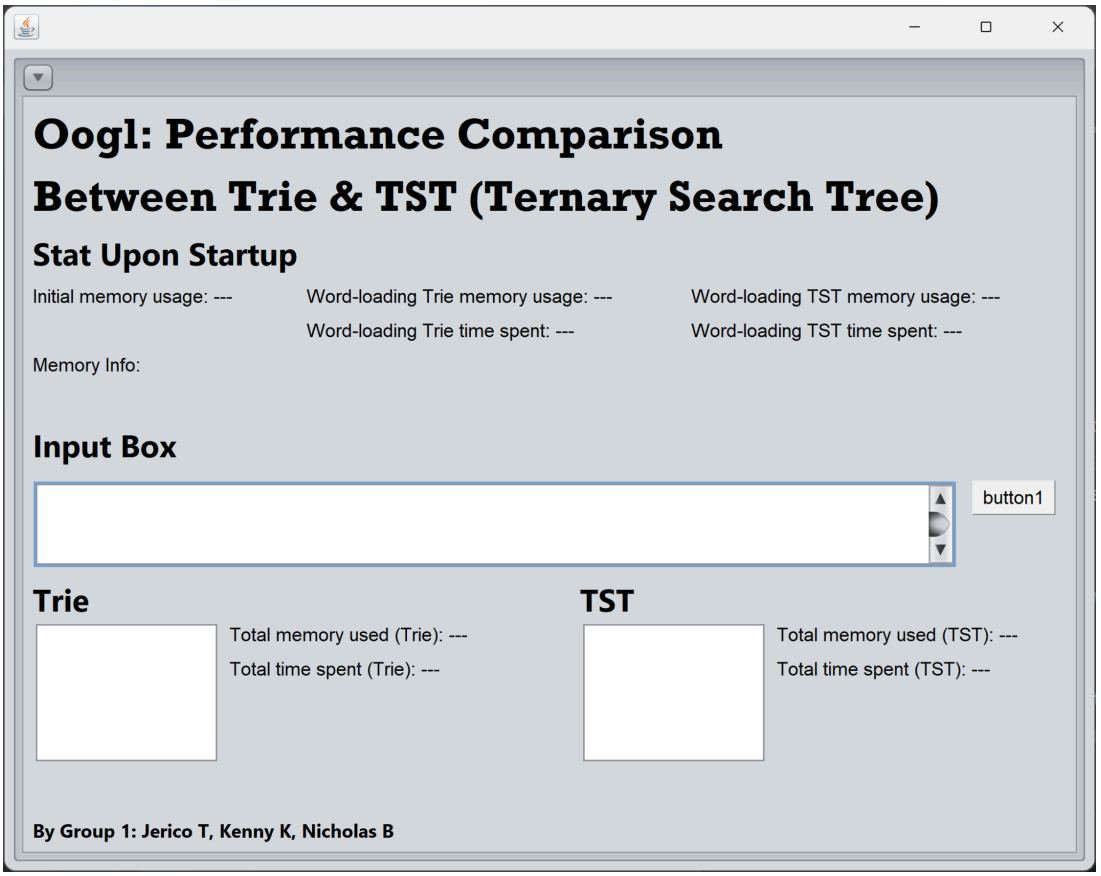
The results also highlight the value of more pragmatic issues like cache behavior and implementation details, which can easily swamp theoretical complexity in practice. The project also notes the possibility of improvement, in particular related to better TST traversal and alternate ways to implement the Trie, including radix trees or other cache-efficient storage methods.

In the end, both data structures demonstrate their distinctive power: Tries are more precise, as their traversal and filtering logic is simple, and TSTs have better run time performance and memory usage. They can both be optimized with nice implementations and structural improvements to make them applicable in the creation of responsive and scalable autocomplete systems.

Chapter 3

Documentation

3.1. Screenshots



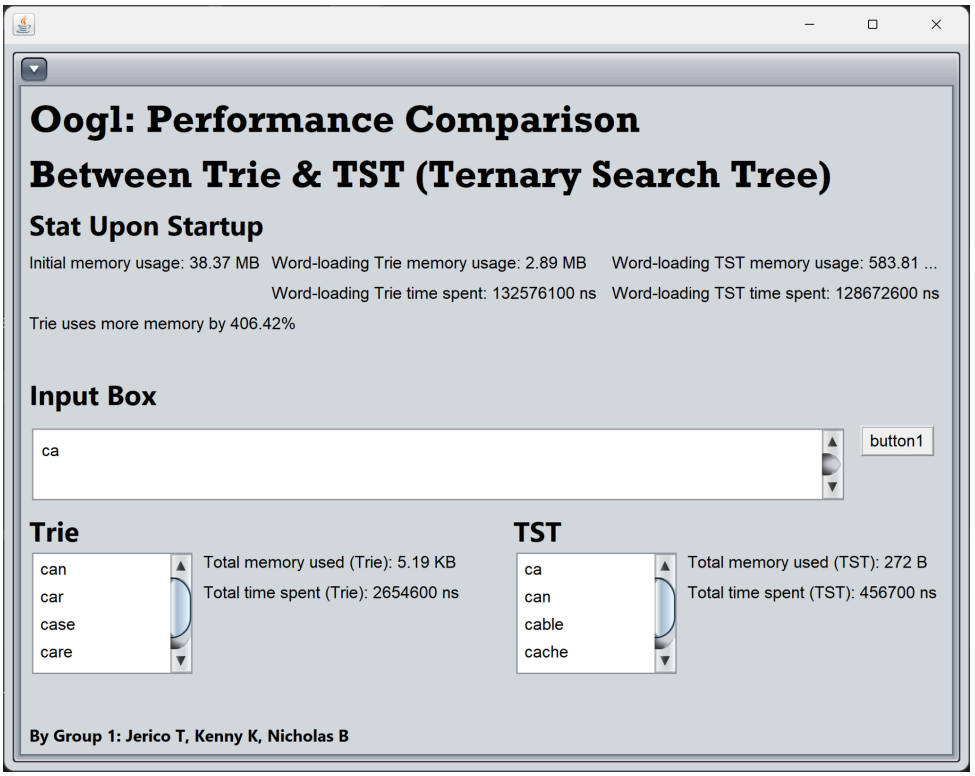
```java
package example;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;

public class TernarySearchTree {
    private TSTNode root;

    public void insert(String word, int frequency) {
        root = insert(root, word, 0, frequency); // Index 0 is a default value.
    }

    /// Recursive helper for insertion...
    private TSTNode insert(TSTNode node, String word, int index, int frequency) {
        if (word.isEmpty()) return node; // Skips over empty words, if any.

        char c = word.charAt(index); // Retrieves the letter from the word at a specified index.

        if (node == null) { // If the current node is not populated, then add the retrieved letter.
            node = new TSTNode(c);
        }

        if (c < node.data) { // If the current letter is smaller than the node's letter.
            node.left = insert(node.left, word, index, frequency);
        } else if (c > node.data) { // If the current letter is bigger than the node's letter.
            node.right = insert(node.right, word, index, frequency);
        } else { // If the letters are the same.
            /// This part usually occurs when:
            /// 1. The word is new, populating the middle branch.
            /// 2. A new word has the same prefix as another word already inserted.
            if (index < word.length() - 1) {
                node.middle = insert(node.middle, word, index + 1, frequency); // If the word is not finished.
            } else {
                node.isEndOfWord = true;
                node.wordFrequency = frequency;
```

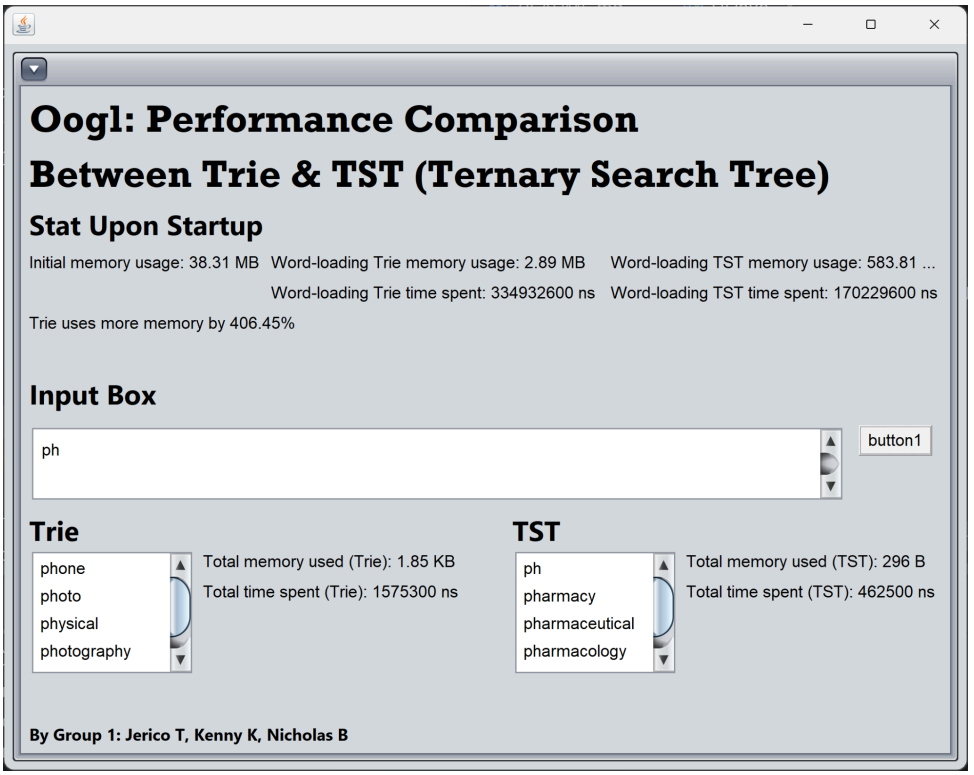
```java
package example;

import java.util.*;

public class Trie {
    private final TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word, int frequency) {
        TrieNode current = root; // Starts at the root node of Trie

        for (char c : word.toCharArray()) {
            current = current.getChildren().computeIfAbsent(c, ch -> new TrieNode()); // Creates a new branch if "c" does not exist
        }
        current.setEndOfWord(true);
        current.setWordFrequency(frequency);
    }

    public List<String> getSuggestions(String prefix, int limit) {
        List<String> suggestions = new ArrayList<>();  // Gets a list of suggestions
        TrieNode prefixNode = findNode(prefix); // Goes to the node with a trace of the appropriate prefix

        /// Goes to each branch sourced from the prefix node.
        /// It traverses each branch until it reaches the end.
        /// Returns the word that has reached the end with their frequency.
        if (prefixNode != null) {
            PriorityQueue<WordFrequency> pq = new PriorityQueue<>(
                    Comparator.comparingInt(WordFrequency::getFrequency).reversed()
            );

            collectWordsWithFrequency(prefixNode, prefix, pq);

            int count = 0;
            while (!pq.isEmpty() && count < limit) {
```

```
package example;

import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import com.opencsv.CSVReader;
import com.opencsv.exceptions.CsvException;

public class Main {
    private Trie trie;
    private TernarySearchTree tst;
    // public static String beforeInitMemory;
    // public static long trieLoadTime;
    // public static long tstLoadTime;
    // public static long trieMemoryUsage;
    // public static long tstMemoryUsage;

    // public static void main(String[] args, String input) {
    //     try {
    //         // Track memory before initialization
    //         System.out.println("===== STARTING MEMORY TRACKING =====");
    //         Runtime runtime = Runtime.getRuntime();
    //         System.gc(); // Request garbage collection to get more accurate memory readings
    //         beforeInitMemory = formatMemorySize(runtime.totalMemory() - runtime.freeMemory());
    //         // System.out.println("Initial memory usage: " + formatMemorySize(beforeInitMemory));


    //         Main autocomplete = new Main("filtered_words.csv");

    //         // Test the autocomplete system
    //         // String input = "prog";

    //         // Memory before Trie suggestion
    //         System.gc();
    //         long beforeTrieSuggestionMemory = runtime.totalMemory() - runtime.freeMemory();
```

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/GUIForms/JFrame.java to edit this template
 */
package example;
import java.util.ArrayList;
import java.util.List;
import java.io.IOException;
import com.opencsv.exceptions.CsvException;

/**
 *
 * @author kenny
 */
public class UI extends javax.swing.JFrame {
    private Trie trie;
    private TernarySearchTree tst;

    /**
     * Creates new form UI
     */

    public UI() {
        initComponents();
    }

    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN:initComponents
    private void initComponents() {

        trie = new Trie();
        tst = new TernarySearchTree();
```

3.2. Evidence of Working Program





Resources

4.1. Resources

1. GeeksforGeeks, “Trie | (Insert and Search),” GeeksforGeeks, Oct. 05, 2011. https://www.geeksforgeeks.org/trie-insert-and-search/
2. “Ternary Search Tree,” GeeksforGeeks, Jan. 13, 2013. https://www.geeksforgeeks.org/ternary-search-tree/
3. “Ternary Search Trees,” Dr. Dobb’s, 2015. https://www.drdobbs.com/database/ternary-search-trees/184410528 (accessed Jun. 1, 2025).
4. “Ternary Search Tries for Fast Flexible String Search: Part 1,” Hackthology.com, Jun. 02, 2011. https://hackthology.com/ternary-search-tries-for-fast-flexible-string-search-part-1.html (accessed Jun. 1, 2025).
5. “Trie Data Structure,” CVR Bioinformatics, Nov. 17, 2014. https://bioinformatics.cvr.ac.uk/trie-data-structure/
6. Askitis, N., & Zobel, J. (2005). Cache-Conscious Collision Resolution in String Hash Tables.
7. T. Janssen, “OOP Concept for Beginners: What is Encapsulation,” Stackify, Mar. 10, 2023. https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/
8. A. Kumar, “What Is Abstraction in OOPS? Definition, Types, and Advantages,” Pwskills, Oct. 30, 2023. https://pwskills.com/blog/what-is-abstraction-in-oops-definition-types-and-advantages/
9. Refactoring Guru, “Facade,” Refactoring.guru, 2014. https://refactoring.guru/design-patterns/facade
10. tutorialspoint, “Design Patterns - Observer Pattern - Tutorialspoint,” www.tutorialspoint.com. https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

4.2 Appendix

- Program Manual
- Link to Git
- Presentation File