Project 3 Group 68 Report

Administrative

Team Name: Project 3 group 68

Members: Aditya Nandakumar(AdityaNandakumar), Jahmal Smith(jahmalsmith)

GitHub Repository: https://github.com/AdityaNandakumar/Project-3-cop3530.git

Youtube Video: https://youtu.be/yz3mO-prsG4

Proposal

1. Problem Statement

The core problem addressed in this project is how to store and manage a large list of strings in a more memory efficient manner without significantly compromising performance. As applications that rely on string based datasets (such as autocomplete systems, spell checkers, and language parsers) continue to grow, the need for data structures that optimize both time and space complexity becomes increasingly important. This project aims to investigate whether alternative data structures, particularly tries and hash tables can offer superior performance or memory usage compared to a standard array based approach.

2. Motivation

Arrays are commonly used to store collections of strings due to their simplicity and direct access by index. However, arrays are inefficient for operations like prefix search and can become memory intensive when dealing with large datasets, especially if there is redundancy or shared prefixes among strings. While hash tables provide fast lookup, they do not support operations such as prefix-based search and can have high overhead due to hashing and collision resolution. Tries, on the other hand, are specifically designed for prefix operations but are often considered memory-hungry due to their node structure. This problem is relevant in contexts where efficiency at scale is critical for example, in search engines, mobile typing interfaces, and genetic data processing. Our motivation is to understand the trade-offs between these three structures in real-world conditions by testing them on a dataset of randomly generated strings. Since we already have a solid understanding of standard arrays, our focus will be on implementing and analyzing tries and hash tables to explore their advantages and limitations

3. Features Implemented

Our project implements the following core features across each data structure:

- Insert: Add a new string to the dataset.
- Search: Check whether a given string exists in the dataset.
- Delete: Remove a string from the dataset if it exists.

These features are implemented in three different ways, corresponding to:

- A hash table (using built-in hashing or a custom hash function),
- A trie (each node representing a character and linked via children).

4. Description of Data

The dataset consists of randomly generated strings, each composed of lowercase English letters, with lengths varying between 5 and 15 characters. The set includes several thousand entries to ensure that memory and performance differences become meaningful. These strings are not designed to resemble dictionary words, which helps avoid giving an unfair advantage to trie compression.

5. Tools, Languages, APIs, and Libraries Used

We are using C++ for the core implementation of all data structures due to its performance, control, and ecosystem support. C++ enables fine-grained memory management, which is crucial for accurately measuring and comparing memory usage across different structures. It also offers high execution efficiency, making it ideal for performance benchmarking. For the user interface, we integrated Qt, a powerful C++ framework that supports cross-platform GUI development. Additionally, the Standard Template Library (STL) provides a solid baseline for comparison with our custom implementations. In addition, we used Balsamiq to create interface visuals and diagrams illustrating how each structure stores the data.

6. Algorithms Implemented

- Trie Operations: Each word is inserted by character, branching into nodes where necessary. Search and delete follow similar logic, with recursive cleanup for deletion.
- Hash Table Operations: Strings are hashed and stored in buckets. We used open addressing with linear probing for collision resolution.
- Array Operations: Strings are stored in a linear array or vector. Insertion appends to the end, search uses std::find(), and deletion uses erasure by value. (Not used but kept in mind during analysis)

7. Additional Data Structures / Concepts

While the core focus is on tries, hash tables, and arrays, we also briefly considered the impact of:

- Prefix trees with compression (radix trees)
- Double hashing as an alternative collision strategy
- Sorted arrays with binary search (as an optimized array baseline)

However, these were excluded from the final analysis to keep the scope manageable and to maintain focus on the primary comparison.

8. Distribution of Responsibility and Roles

- Aditya:
    - Implementing the data structures in C++
    - Designing and executing performance tests
    - Creating Youtube video
    - Creating GitHub repository
- Jahmal:
    - Writing and organizing the report

Analysis

The only major change to the project implementation was the integration of a user interface using the Qt framework. This allowed us to present the data structures and their operations (insert, search, delete) in a more interactive and visually accessible way. The core logic, including the structure and performance of the underlying data models (Trie and Hash Table), remained unchanged. This ensured a fair comparison between the implementations.

In analyzing performance, we focused on three fundamental operations: search, insert, and delete. Below is a breakdown of the expected time complexities and practical observations for each data structure:

Search Operation:

- Trie:
  The time complexity of searching in a trie is O(n), where n is the length of the input string. This is because the trie processes each character sequentially from the root. Despite the linear dependency on string length, the search is generally very fast in practice due to minimal overhead per character and early exits for

non-existent prefixes.

- Hash Table:
  In the best case, search is O(1), thanks to direct access via the hash function. However, in the worst case such as when many keys collide into the same bucket search degrades to O(n), where n is the number of elements in that bucket (in open addressing or chaining scenarios). With a good hash function and load factor management, worst-case behavior is rare.

Insert Operation:

- Trie:
  The worst-case time complexity for insertion is O(n), where n is the length of the string being inserted. This occurs when every character in the string is new and requires creating a new node for each character along the path from the root. This is common when the string shares no prefix with any existing entry in the trie.

- Hash Table:
  In the best case, search is O(1), thanks to direct access via the hash function. However, in the worst case such as when many keys collide into the same bucket search degrades to O(n), where n is the number of elements in that bucket (in open addressing or chaining scenarios). With a good hash function and load factor management, worst-case behavior is rare.

Delete Operation:

- Trie:
  The worst-case time complexity for deletion is O(n), where n is the length of the string. This scenario happens when the string is unique and all of its characters must be traversed and potentially deleted. Deleting such a string requires checking each node up the path to determine whether it is shared with other strings or can be safely removed.

- Hash Table:
  The worst-case deletion time is O(n). This can occur in cases of severe hash collisions, where many keys map to the same bucket. Deleting a key then involves scanning through all entries in that bucket (or all probe sequences) to find and remove the target element.

Reflection

Learning Qt for the first time presented several challenges. Coming from a background with little or no prior exposure to Qt's signal-slot mechanism, UI layout system, and event-driven architecture, the initial learning curve was steeper than expected. Understanding how widgets interact, how to properly manage layout hierarchies, and how to connect UI elements to backend logic using signals and slots required both reading documentation and experimenting through trial and error.

In hindsight, I might have chosen a different UI toolkit such as SDL (Simple DirectMedia Layer) or even a lightweight game engine like Unity or Godot, depending on the project's goals. These alternatives may have provided a more familiar or intuitive development environment, particularly for graphical or interactive applications. SDL, for instance, offers more low-level control over rendering and input, which could have simplified certain aspects of UI design, especially if a more custom or minimal interface was acceptable.

That said, using Qt also brought several advantages. It's a robust and well-documented framework, and once I overcame the initial complexity, I gained a much better appreciation for its powerful layout management, cross-platform capabilities, and extensibility. The experience of working with Qt has deepened my understanding of GUI application development and given me skills that will be useful in future desktop software projects. Despite the challenges, I'm ultimately glad I took the opportunity to learn it.

References

https://www.geeksforgeeks.org/dsa/trie-insert-and-search/

https://balsamiq.cloud/sh0yb47/pitjiso