COLORADO MESA UNIVERSITY

Final Assignment Due: 11:59 PM May 10th, 2025

CSCI 250 Introduction to Algorithms

By: McKenzie Swindler and Caleb Bishop

Aim

The aim of this assignment is to practice building and using Hash tables (chaining and linear probing), Rabin- Karp pattern matching, and resizable arrays.

Task Description

The task involves testing pattern matching algorithms using a stream dataset. The provided dataset is a text file named "A Scandal In Bohemia.txt", which contains twelve works by Sir Arthur Conan Doyle. The assignment requires developing a suitable hash-table implementation with open hashing (chaining) and Linear probing to store the contents of the text file. Specifically, works I through VI should be stored using open hashing, while works VII to XII should be stored using Linear Probing. The data should be read only once as a stream, i.e., one string at a time. Once the data is processed, the program should display and record to a file a list of word occurrences from highest to lowest and vice versa of the 80 most and least repeated words (and their count) in Conan Doyle's works. It's important to note that capitalization should not matter (i.e., "Watson" and "watson" should be counted as the same word), but hyphenation is critical, and double hyphens "–" should be discarded. As the program starts processing work IX "The Adventure of the Engineer's Thumb", it should prompt the user for search keys (up to 8, delimited by "@@@" if less than 8) and display the position of each key's occurrence in the text (i.e., word count position). It's essential not to assume that any of the keys sought are unique, so each occurrence of each key must be accounted for. Rabin-Karp pattern matching algorithm and Horner's rule for the rolling hash should be utilized for this purpose.

Key Reports

The assignment requires reporting on several key aspects:

1. The occupancy ratio ro be used in linear probing. This involves experimenting with different values such as 50%, 70%, and 80%, and reporting runtimes in nanoseconds.

2. Optimizing chain length in open hashing. At least three experiments should be conducted, and runtimes in nanoseconds should be reported.

3. Experimentation with different hash functions. A simple function such as f(r) = r%hsize should be the initial attempt.
4. Handling collisions in the table for linear probing. The collision resolution method implemented must be described, with research and inclusion of a method described in the lecture.
5. The necessity of an interface file (a ".h" file) for the functions implemented.
6. Writing a function to prompt a user for a word, display the number of occurrences of this word in the text, and the locations of said occurrences in "The Adventure of the Engineer's Thumb".
7. Implementing a function to output a list of the 80 least frequently occurring words in the text.
8. Implementing a function to output a list of the 80 most frequently occurring words in the text.
9. A function to output the number of sentences in the text.
10. Reporting the runtime for each task, while keeping reusability in mind.

Experiments

We did three experiments using different load factors for linear probing, varying table sizes for chaining, and comparing hash functions. For the different load factors (50%, 70%, and 80%) the load factor of 70% was the fastest. When using different table sizes there was more of a difference. The largest table size had the fastest runtime and as the table sizes got smaller the runtime got larger. Although larger tables consume more memory, they reduce collision frequency and improve average-case time complexity for insertions and lookups. The last experiment was between different hash functions. We used Horner's method and a simple mod hash. Horner's rule consistently outperformed the simpler mod-based hash function due to its superior distribution, resulting in fewer collisions and faster lookups. This highlights the importance of investing in a quality hash function even if it adds slight computational overhead. Overall, the results align well with theoretical expectations and reinforce best practices in designing efficient hash table implementations.

```
=== Running Experiments ===

=== Experiment 1: Linear Probing with Varying Load Factors ===
Load factor: 0.5 → Average Time (10 runs): 4245356 ns
Load factor: 0.7 → Average Time (10 runs): 3552384 ns
Load factor: 0.8 → Average Time (10 runs): 3767532 ns

=== Experiment 2: Chaining with Varying Table Sizes ===
Table size: 5003 → Average Time (10 runs): 4051559 ns
Table size: 10007 → Average Time (10 runs): 3673903 ns
Table size: 20011 → Average Time (10 runs): 3461823 ns

=== Experiment 3: Comparing Hash Functions (Chaining) ===
Horner's → Average Time (10 runs): 4915917 ns
Simple mod hash → Average Time (10 runs): 7355605 ns

=== Experiment 4: Collision Handling (Linear Probing) ===
Collision resolution uses linear probing: if a collision occurs, probe the next slot using (i + 1) % hsize.
This method is based on open addressing, as discussed in class.
```

Whitelisted Libraries

The allowed libraries for this assignment are “iostream”, “cassert”, “iomanip”, “fstream”, “chrono”, “cstdlib”, “ctype”, and “(c)string”, STL containers, STL iterators, no STL algorithms.

Implementation

The main file, FinalAssignment.cpp, should be created with the appropriate entry in the makefile. This program, when built, will expect a command-line argument consisting of an input text file and an output text file name. It will read the data in the input file (strings or cstrings) and write to the output file the outcomes of all tasks listed above, with clearly labeled sections and runtimes for each task. The output strings should all be in lowercase.

Testing

The functionality of the program should be tested by augmenting the driver program from previous steps in FinalAssignment.cpp to form a menu driver to test all tasks. A “zero” menu option request should terminate the program.