CSCI Assignment 4 Report
Hash Tables, Pattern Matching, Resizable Arrays

Paul Connett, Chica Gomes, Gabe Medlock

May 9, 2025

1 File Interactions

DynamicArray.cpp/h Custom self-resizing array with doubling strategy.

Chaining.cpp/h Implements a hash table using separate chaining. Rehashing occurs when average chain length exceeds a target threshold.

Linear.cpp/h Implements a linear-probing hash table. Rehashing is triggered when load factor exceeds 0.50, 0.70, or 0.80.

RabinKarp.cpp/h Rolling-hash pattern matcher using base 256 and prime 101. Returns all match indices.

Pattern.cpp/h
- Ingests and cleans text, routes insertion to the appropriate table based on work number.
- Generates frequency lists of the 80 most and least frequent words.
- Provides search and sentence-counting functionality.

Logger.cpp/h Time-stamped logger that appends messages to `run.log`.

FinalAssignment.cpp Menu-driven interface that calls `Pattern`, `RabinKarp`, and logs runtimes.

2 Algorithm Details

2.1 Hash Functions

We used a simple hash function where we add up the characters in a word (converted to numbers) and then take the remainder when dividing by the table size. This spreads words across the table so they land in different spots.

We also tested a different method called a multiplicative hash. It multiplies the number version of the word by a constant (2654435761) and uses part of the result. This method can sometimes reduce clustering better than just adding up characters.

Both of these methods were used to convert words into numbers that map to table positions, which is how we place and find them in the hash table.

2.2 Rehash Policies

Chaining: We rehashed the table when the average chain length exceeded 2, 4, or 8. **Linear Probing**: We rehashed when the load factor exceeded thresholds of 0.50, 0.70, or 0.80.

2.3 Rabin–Karp

We implemented Rabin–Karp using a rolling hash with Horner's rule:

$$H_{i+1} = \big(b(H_i - T_i b^{m-1}) + T_{i+m}\big) \bmod p$$

with $b = 256$, $p = 101$, and H_0 as the initial hash. This results in worst-case time $O(nm)$, and expected time $O(n+m)$ under uniform random input.

3 Results

3.1 Runtime Measurements

Operation	**Average Runtime (ns)**
Build chaining (avg chain 2)	2,093,837,946
Build chaining (avg chain 4.18)	1,887,300,294
Build chaining (avg chain 8.36)	1,671,559,492
Build linear (load 0.50)	1,728,503,001
Build linear (load 0.70)	1,923,945,158
Build linear (load 0.80)	1,844,269,463
80 most frequent word list	3,424,706,805
Rabin–Karp (8 patterns)	10,456,875,530

Table 1: Selected empirical timings from our implementation

Linear Probing Occupancy Test

We tested three different load factors and recorded average insertion time:

Occupancy	Insert Count	Table Size
50%	95	190
70%	95	136
80%	95	119

Equation Used:

$$\text{Load Factor} = \frac{\text{Insert Count}}{\text{Table Size}}$$

Chaining Chain-Length Performance

We tested chaining performance under different average chain lengths:

- Chain Length 2 $\rightarrow$ 500-slot table, 1,043 inserts
- Chain Length 4.18 $\rightarrow$ 100-slot table, 418 inserts
- Chain Length 8.36 $\rightarrow$ 50-slot table, 418 inserts

Equation Used:

$$\text{Average Chain Length} = \frac{\text{Insert Count}}{\text{Table Size}}$$

Sentence Count

Sentence count: 5255