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 (doubling capacity) supporting push/pop, random access, copy, clear, `shrink_to_fit()`.

`Chaining.cpp/h` Separate-chaining hash table. Chains stored as linked lists. Rehashing on average chain-length thresholds.

`Linear.cpp/h` Open-addressing hash table with linear probing. Rehash when the load factor exceeds specified thresholds.

`RabinKarp.cpp/h` Rolling-hash pattern matcher (base 256, prime 101) implementing Horner's rule; returns all match indices.

`Pattern.cpp/h`
- Stream-reads Conan Doyle's twelve works, cleans & lowercases words, preserves single hyphens, discards double hyphens.
- Routes works I–VI into chaining table, VII–XII into linear table.
- Builds list of word frequencies, outputs 80 most/ 80 least frequent.
- Captures all words of "Work IX" for on-demand Rabin–Karp search.
- Counts sentences by terminal punctuation.

`Logger.cpp/h` Simple time-stamped logger writing to `logger.txt`.

`FinalAssignment.cpp` Menu-driven driver invoking `Pattern`, logs runtimes for each task.

2 Algorithm Details

2.1 Hash Functions

Multiplicative hash:

$$h'(w) = \Big(\mathrm{crc32}(w) \times 2654435761\Big) \gg \big(32 - \log_2 N\big)$$

We compared both; the simple additive hash gave a uniform spread with minimal overhead, so it was retained.

2.2 Rehash Policies

- **Chaining:** Rehash to double capacity when average chain length exceeds thresholds (2, 4, 8).
- **Linear Probing:** Rehash when load factor (elements/table size) exceeds thresholds (0.50, 0.70, 0.80).

2.3 Collision Resolution

Chaining: New entries prepended to each bucket's linked list. **Linear Probing:** On collision, probe sequence $(h + k) \bmod N$ for $k = 1, 2, \ldots$.

2.4 Interface Files

Each module has a header ('.h') exposing only its API. This hides implementation details, enables recompilation-minimal changes, and encourages reuse.

2.5 Rabin–Karp Pattern Matching

Rolling-hash via Horner's rule:

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

with $b = 256$, $p = 101$; runs in $O(n + m)$ expected time.

3 Results

3.1 Runtime Measurements

Operation	**Avg 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

3.2 Linear Probing Occupancy Test

$$\text{Load factor} = \frac{\text{Insert count}}{\text{Table size}}$$

Load %	Inserts	Table Size
50%	95	190
70%	95	136
80%	95	119

3.3 Chaining Chain-Length Performance

- Avg chain 2 (size 500) → 2 093 837 946 ns
- Avg chain 4.18 (size 100) → 1 887 300 294 ns
- Avg chain 8.36 (size 50) → 1 671 559 492 ns

$$\text{Avg chain length} = \frac{\#\text{inserts}}{\text{table size}}$$

3.4 Sentence Count

Detected sentences: **5255**

4 Discussion & Insights

- *Chaining vs. Probing:* Short chains outperformed probing at medium loads; probing degraded near high loads due to clustering.
- *Hash Function:* Additive hash proved adequate; multiplicative offered no clear benefit for this dataset.
- *Modularity:* Clear header/API separation allowed easy swapping of collision strategies and hash functions.
- *Pattern Matching:* Rabin–Karp efficiently located multiple occurrences in “Work IX” with minimal extra memory.