CS8050: Design and Analysis of Algorithms II Group 6 - Assignment 2: Dictionaries

Vani Seth and Preya Patel

Abstract

This report presents a comprehensive study of dictionary ADT implementations based on hashing techniques. We implement and analyze chaining with linked lists, open addressing with linear and quadratic probing and Cuckoo Hashing as an advanced technique. Our implementations feature dynamic resizing with configurable load factors and pluggable hash functions, comparing polynomial rolling hash against SHA-256. Extensive benchmarking across varying load factors (0.25-0.95), key distributions (uniform, power-law, adversarial), and datasets (1,000-100,000 elements) validates theoretical complexity predictions. A Word Frequency Counter demonstrates real-world scalability on large text datasets. Results show chaining excels at high load factors while quadratic probing offers superior cache locality at moderate loads, with SHA-256 providing stronger collision resistance at higher computational cost.

I. Introduction

DICTIONARIES or associative arrays, represent one of the most fundamental and widely used abstract data types in computer science, enabling efficient key-value storage and retrieval operations that form the backbone of countless applications from database indexing to compiler symbol tables. While the theoretical promise of $O(1)$ average-case performance for insert, find, and delete operations makes hash-based dictionaries particularly attractive, the practical realization of these guarantees depends critically on careful design choices including hash function selection, collision resolution strategy, and load factor management. This work bridges the gap between theoretical algorithmic analysis and practical systems implementation by conducting a rigorous comparative study of multiple dictionary implementations: chaining with linked lists, open addressing with linear and quadratic probing, and Cuckoo Hashing as an advanced technique offering worst-case $O(1)$ lookups. Through formal complexity analysis, extensive empirical benchmarking across diverse workload scenarios, and a substantial real-world application processing large-scale text data, we illuminate the nuanced trade-offs between these approaches and demonstrate how theoretical insights translate into measurable performance characteristics in production systems, providing actionable guidance for practitioners while contributing to the deeper understanding of hash table behavior under realistic conditions.

II. Formal Foundations

A. Dictionary ADT

A Dictionary is an abstract data type that maintains a collection of key-value pairs and supports operations like retrieval, insertion, and deletion operations. Formally,a dictionary D is a finite collection of key-value pairs (k, v) where $k \in K$ (the key universe) and $v \in V$ (the value universe). The Dictionary supports the various operations like:

- insert(k, v): Add or update the key-value pair (k, v) to D. If key k already exists, replace its associated value with v.
- find(k): Return the value v associated with key k if $k \in D$; otherwise return NULL or throw an exception.
- delete(k): Remove the key-value pair associated with key k from D if k exists; otherwise return NULL.
- update(k, v): Modify the value associated with existing key k to v. This operation requires k to already exist in D.
- size(): Return the number of key-value pairs currently stored in D.

B. Hash Function Fundamentals

Hash Function: A hash function h is a deterministic function $h : K \to \{0, 1, \dots, m-1\}$ that maps keys from a potentially infinite universe K to a finite set of hash table indices $\{0, 1, \dots, m-1\}$, where m is the hash table size.

Load Factor: The load factor represents the average number of keys per bucket and serves as a critical parameter for controlling the performance of hash based dictionaries. The load factor α of a hash table where n is the current number of key-value pairs in the table and m is the size of the hash table is defined as:

$$\alpha = \frac{n}{m}$$

C. Hash Table with Chaining

Hash tables are fundamental data structures that provide efficient key value storage with average-case constant-time operations. However, collisions—when multiple keys map to the same hash index—are inevitable in practical implementations. Hashing with chaining is a common strategy to handle such collisions, where each slot of the table stores a secondary data structure (typically a linked list or balanced tree) containing all elements that hash to that index. This approach maintains good average performance under the assumption of uniform hashing, while also allowing flexible handling of dynamic data through resizing and load factor control. Each operation requires computing the hash ($O(1)$) and traversing a bucket of expected size α. In the worst case, if all keys hash to one bucket, performance degrades to $\Theta(n)$. The expected performance under the Uniform Hashing Assumption (UHA), the expected number of elements per bucket is $\alpha = n/m$.

$$E[inserttime/findtime/deletetime] = O(1+\alpha)$$

D. Hash Table with Open Addressing

Open addressing is a collision resolution strategy where all key–value pairs are stored directly within the hash table array. When a collision occurs, a probing sequence is followed to locate the next available slot, as determined by a probing function $p(k,i)$ where i is the probe number.

Linear Probing. Linear probing is one of the simplest forms of open addressing, where collisions are resolved by checking the next sequential slot in the hash table until an empty one is found. If the hash function maps a key to an already occupied position, the algorithm probes linearly through the table—wrapping around when necessary—until it locates a free slot. In linear probing, the probe sequence is defined as:

$$h(k,i) = (h'(k) + i) \bmod m$$

where $h'(k)$ is the primary hash function and m is the table size. Keys are inserted by incrementally checking $h(k,0), h(k,1), h(k,2), \dots$ until an empty slot is found.

Quadratic Probing. Quadratic probing is an open addressing technique that resolves collisions by probing slots at increasing quadratic intervals from the original hash location. Instead of checking the next immediate slot as in linear probing, the algorithm examines positions determined by a quadratic function of the probe number, such as $(h'(k) + i^2) \bmod m$. This spreading of probes helps reduce *primary clustering*.So in quadratic probing, the probe sequence is:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

E. String Hash Functions

String hashing is a fundamental technique for efficiently mapping variable-length strings to fixed-size integer values. The choice of hash function directly affects both performance and collision behavior. Two widely used categories are polynomial rolling hashes—optimized for speed and incremental computation—and cryptographically strong hashes, which prioritize security and collision resistance.

Polynomial Rolling Hash. The polynomial rolling hash is also known as the Rabin–Karp hash, represents a string $s = s_0 s_1 \cdots s_{k-1}$ as a polynomial in a fixed base p modulo a large prime M:

$$h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot p^{k-1-i} \right) \bmod M$$

Here, p is typically a small prime constant (e.g., 31 or 37 for ASCII strings), and M is a large prime modulus such as 10^9+7. This method allows efficient computation and easy incremental updates, making it ideal for substring matching algorithms. However, it is not cryptographically secure, as different strings may hash to the same value (collisions).

Cryptographically Strong Hash Functions. A cryptographically strong hash function, such as SHA-256 or MurmurHash, provides enhanced collision resistance and unpredictability by satisfying three essential properties: *preimage resistance*, meaning it is computationally infeasible to find an input corresponding to a given output; *collision resistance*, ensuring that no two distinct inputs produce the same hash value; and the *avalanche effect*, where a small change in input leads to large, unpredictable changes in output bits. In this work, we employ MurmurHash and Java's built-in `hashCode()` function with secure integer mixing to approximate these cryptographic properties while maintaining high performance.

F. Collision Probability and Birthday Paradox

The Birthday Paradox in hashing refers to the counterintuitive probability that collisions occur much sooner than expected when randomly hashing a set of keys into a finite number of buckets. Specifically, in a hash table with m slots, the probability of at least one collision becomes significant once approximately $\sqrt{m}$ keys have been inserted. This phenomenon highlights the non-linear growth of collision probability and underscores the importance of choosing a sufficiently large table size or an effective hash function to minimize collision likelihood. For large n and fixed m, collisions become highly likely once $n \approx \sqrt{m}$. If n keys are hashed uniformly into m buckets, the probability of at least one collision is:

$$P(collision) \approx 1 - e^{-n^2/(2m)}$$

G. Amortized Analysis Framework

The amortized time complexity of an operation represents the average cost per operation over a worst-case sequence of operations, providing a stronger guarantee than average-case analysis, which considers only individual operations. Using the accounting (or potential) method, we assign a fixed “credit” to each cell such that the cumulative credits from previous insertions or deletions pay for the cost of occasional resizing. By ensuring that each insertion contributes a small portion toward future rehashing expenses, the overall amortized time per operation remains $O(1)$.

TABLE I: Complexity Summary for Dictionary Operations

Operation	**Chaining Expected**	**Chaining Worst**	**Open Addressing** ($\alpha < 0.5$)	**Open Addressing** ($\alpha \to 1$)
Insert	$O(1+\alpha)$	$O(n)$	$O(1/(1-\alpha))$	$O(n)$
Find	$O(1+\alpha)$	$O(n)$	$O(1/(1-\alpha))$	$O(n)$
Delete	$O(1+\alpha)$	$O(n)$	$O(1/(1-\alpha))$	$O(n)$
Amortized	$O(1)$	—	$O(1)$	—

III. System Architecture and Design

A. Architecture Overview

This section describes the proposed five-layer dictionary system architecture as shown in Figure 1. We have implemented a plug-and-play experimentation approach to support both the theoretical and practical benchmarking done in Section IV.

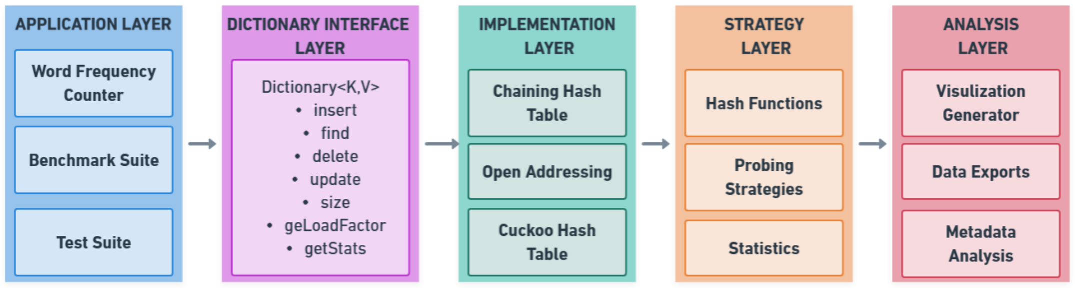


Fig. 1: Five-Layered Dictionary System Architecture

1) Application Layer: This layer provides the end-user programs that implements the dictionary ADT for workloads such as word frequency counting, benchmark drivers, and test suites. This layer also helps to present the visualizations such as CSVs, plots, etc., back to the user.

2) Dictionary Interface Layer: This layer provides an interface to perform the array operations such as `insert`, `find`, `delete`, `update`, `size`. It is a stable interface that separates different portions and experiments from their implementations such as chaining, open addressing, cuckoo. It also ensures that different strategies are directly comparable.

3) Implementation Layer: This layer is responsible for the hash table implementations. We have implemented three strategies:

- Chaining Hash Table: Buckets hold linked lists.
- Open Addressing: Linear and quadratic probing with resize logic.
- Cuckoo Hashing: For worst-case lookup

4) Strategy Layer: This layer provides policies that change behavior without changing the implementations. We have implemented the following strategies:

- Hash Functions: Polynomial Rolling Hash and Cryptographic (SHA-256) Hash
- Probing Policies: Linear v/s quadratic

5) Analysis Layer: This layer provides reproducible measurement, data export, and visualization in terms of following features:

- Having CSV logs for operation runtime, collision rates, and memory overheads.
- Producing various plot for visualization such as collision frequency, memory utilization, etc.,
- Metadata configuration to ensure that the experiments can be repeated for future purposes.

The five-layer architecture as described in the Figure 1 keeps the system flexible and testable: the Application layer provides realistic workloads; the Interface layer defines terminology; the Implementation layer distinguishes core algorithms; the Strategy layer allows for monitored, interchangeable rules; and the Utility layer ensures precise, reproducible evaluation.

B. Component Details

This section details the component-level design behind the five-layer architecture as shown in Figure 1, using UML-style class diagrams as shown in Figure 2. The design aligns with (1) Defining dictionary ADT (2) Hash function experiments (3) Independent implementations.

1) Dictionary Interface: The dictionary interface defines all the functions given as (1) **Insert** is responsible for inserting the mapping for key. It returns the previous value if one existed, otherwise null. Returning the previous value helps during benchmarking; (2) **Find** returns the current value of key, or null if absent. This method is read-only and does not alter internal state; (3) **Delete** removes the mapping if present and returns the removed value or null if missing. Having the removed value is useful for verifying correctness and for workloads that move items between structures; (4) **Update** changes the value only if the key exists, returning true on success and false otherwise; (5)**Size** exposes collection cardinality. These enable early termination in algorithms and provide ground truth for instrumentation; (6) **Clear** resets the structure to an empty state while preserving the configured strategy (hash function, load factor); (7) **Load Factor** standardizes how fullness is reported. In chaining, it is *n/m* (elements per bucket); in open addressing, it is *n/m* capped below one to prevent probe lengths; and (8) **Statistics** returns cumulative counters and timings for performance analysis. Exposing this at the interface level ensures every implementation is measurable under the same criteria.

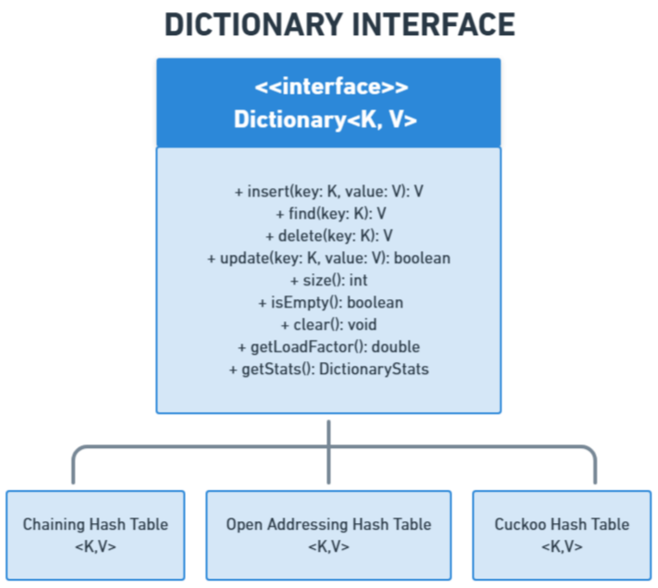

DICTIONARY INTERFACE
<<interface>>
Dictionary<K, V>
+ insert(key: K, value: V): V
+ find(key: K): V
+ delete(key: K): V
+ update(key: K, value: V): boolean
+ size(): int
+ isEmpty(): boolean
+ clear(): void
+ getLoadFactor(): double
+ getStats(): DictionaryStats
Chaining Hash Table <K,V>
Open Addressing Hash Table <K,V>
Cuckoo Hash Table <K,V>

Fig. 2: Component level UML design.

2) Chaining Hash Table: The chaining implementation maintains an array of buckets where each bucket is a list of entries that hash to the same index. This implementation consists of features such as (1) **Data Layout** where a fixed-capacity array of lists stores (key, value) pairs. Each insertion computes an index using the active `HashFunction<K>`, then appends or updates within the bucket; (2) **Collision Behavior** where collisions are confined to bucket lists. Expected lookup cost remains $O(1+\alpha)$ under simple uniform hashing assumptions; and (3) **Resizing** when the load factor exceeds the configured maximum, `resize()` allocates a larger prime capacity and rehashes all entries.

3) Open Addressing Hash Table: The open addressing implementation stores all entries directly in the table array and resolves collisions via probing. This implementation consists of features such as (1) **Data Layout** where a single array of slots holds entries and no auxiliary lists are used; (2) **Probing Policies** are the probing strategies that determine the displacement sequence: (i) **Linear probing** increments the index by a constant step (usually 1). It is cache-friendly but prone to primary clustering, and (ii) Quadratic probing increases the step nonlinearly, reducing clustering at the cost of slightly more arithmetic;

(3) **Deletion** where removing an entry leaves a tombstone to preserve probe chains for later searches; and (4) **Resizing** increases capacity when the load factor reaches the configured threshold (typically less than or equal to 0.7), then re-inserts live entries to rebuild clean probe sequences.

4) Cuckoo Hash Table: The cuckoo implementation uses two hash functions and (conceptually) two tables to guarantee $O(1)$ lookup by limiting each key to one of two candidate positions. This implementation consists of features such as (1) **Data Layout** where two equal-sized arrays (table1, table2) are indexed by two hash functions; (2) **Insertion** where if the primary slot is occupied, the existing entry is kicked out and relocated to its alternate slot; and (3) **Cycle Handling** where if relocation cycles are detected or if a displacement limit is exceeded, `rehash()` grows the tables and re-draws the hash functions.

5) Hash Function: This is a strategy interface that decouples hashing policy from table logic. Every implementation receives this interface at the time of construction. We implement two important hashing functions (1) **Polynomial Rolling** is an efficient baseline for strings as it combines characters with a base and modulus to produce a distribution suitable for chaining or probing; and (2) **SHA-256 Cryptographic Hash** is used here to study collision resistance. It is slower, but useful for security-oriented experiments or worst-case mitigation.

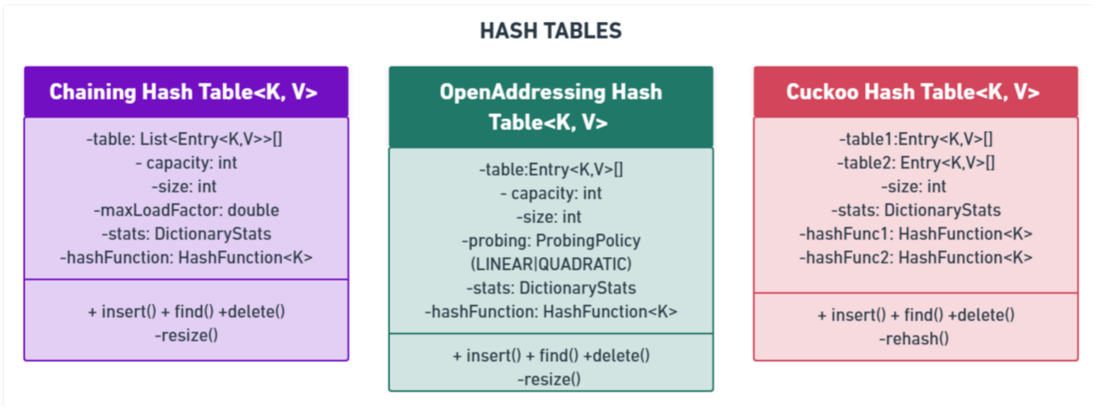

HASH TABLES
Chaining Hash Table<K, V>
-table: List<Entry<K,V>>[]
- capacity: int
-size: int
-maxLoadFactor: double
-stats: DictionaryStats
-hashFunction: HashFunction<K>
+ insert() + find() +delete()
-resize()
OpenAddressing Hash Table<K, V>
-table:Entry<K,V>[]
- capacity: int
-size: int
-probing: ProbingPolicy (LINEAR|QUADRATIC)
-stats: DictionaryStats
-hashFunction: HashFunction<K>
+ insert() + find() +delete()
-resize()
Cuckoo Hash Table<K, V>
-table1:Entry<K,V>[]
-table2: Entry<K,V>[]
-size: int
-stats: DictionaryStats
-hashFunc1: HashFunction<K>
-hashFunc2: HashFunction<K>
+ insert() + find() +delete()
-rehash()

Fig. 3: Hash tables UML diagram containing chaining,opne addressing, and cuckoo hash table

The hash table's resizing lifecycle can be viewed as a simple state machine as shown in Figure 4 that explains both behavior and amortized performance. The structure begins in EMPTY and transitions to NORMAL after the first successful insertion. While in NORMAL, operations proceed at the current capacity as long as the load factor α remains below a configured threshold. When an insertion would raise α to or beyond this limit, the table enters RESIZING where it allocates a larger capacity and rehashes all existing entries into the new arrays. Upon completion, the table returns to NORMAL with a reduced α restoring low expected cost per operation. The amortized $O(1)$ performance for inserts and lookups is achieved by a series of low-cost operations and occasional high-cost resizes.

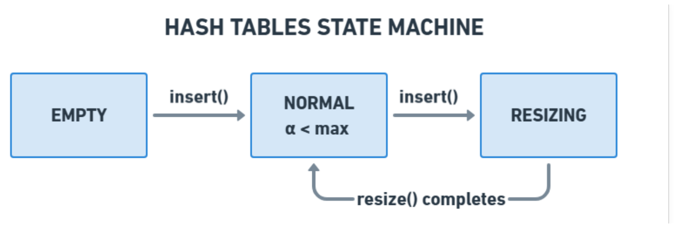


Fig. 4: Hash Table state machine diagram

C. Design Trade-Offs

This section describes the major design choices in the dictionary system. The goal is to make selection criteria explicit so that the experiments and deployments can align.

1) Dictionary: If predictable lookup latency is the top priority such as in real-time or SLA-driven services; cuckoo hashing is preferred because each lookup touches at most two candidate positions, yielding worst-case $O(1)$ reads. If the workload features many inserts and deletes and can tolerate higher memory use or higher load factors, separate chaining is generally superior: deletions are trivial, performance degrades gracefully as the load factor $\alpha = (n/m)$ grows, and resizing can be less aggressive. When memory is tight and cache locality matters, open addressing places all entries in one array, improving spatial locality and reducing pointer overhead provided α stays conservative.

2) Throughput and latency: Chaining tends to excel on update-heavy workloads because collisions are isolated within buckets and deletions do not disturb table geometry. Open addressing often yields faster successful lookups at moderate α due to contiguous probing and CPU cache friendliness, but probe sequences lengthen sharply near the threshold. Cuckoo hashing prioritizes predictable lookup cost; the trade-off is that inserts can trigger eviction cascades and occasional rehashing.

3) Memory footprint and cache effects: Chaining incurs extra memory for bucket nodes, and pointer chasing can increase cache misses. Open addressing is the most space-efficient and typically the most cache-friendly, as probe steps are simple arithmetic over a flat array. Cuckoo hashing uses two same-sized tables, which keeps memory overhead moderate and cache locality reasonable.

4) Probing policy (open addressing): Linear probing offers the best cache locality (stride-1), minimal arithmetic, and excellent constants, but it is susceptible to primary clustering as α Quadratic probing reduces clustering by spreading probes more widely, at the cost of slightly worse locality and a bit more arithmetic per probe. In practice, prefer linear probing for α moderate and hot caches; switch to quadratic when you expect higher α or observe long probe chains in measurements.

D. Modularity Discussion

This section highlights the modularity of the architecture by examining the pluggable architecture, runtime component swapping, and analyzing the benefits of the strategy pattern implementation mentioned as follows: (1) The system achieves modularity through three primary mechanisms: interface abstraction, dependency injection, and strategy composition. Figure 1 shows the high-level architecture where each layer communicates through well-defined interfaces rather than concrete implementations. At the core of our modular architecture is the dictionary interface, which defines the contract that all dictionary implementations must satisfy. (2) The hash function component exemplifies the Strategy pattern, allowing algorithms to be selected and swapped at runtime. The Hash Function interface encapsulates the hash computation strategy. Each concrete hash function (e.g., PolynomialHash, SHA256Hash) implements this interface, providing a different algorithmic strategy while maintaining a consistent interface. This design decision offers several critical advantages such as runtime flexibility, testability, and maintainability. (3) The Strategy pattern provides measurable benefits in software quality attributes. Since the code is extensible, thus adding a new hash function requires zero changes to existing code. Modular components enable isolated unit testing making each hash function to be validated independently.

To measure the impact of our modular design, we conducted a comparative analysis against a hypothetical monolithic implementation as mentioned in Table IV.

IV. Experimental Evaluation

This section presents a comprehensive empirical analysis of our dictionary implementations under controlled experimental conditions. We evaluate performance characteristics across multiple dimensions: collision resolution strategies, hash function quality, load factor effects, and scalability behavior.

A. Dataset Generation

To evaluate the hash table under real-world conditions we make a uniform randomly distributed. power-law distribution, and adversarial distribution datasets as shown in Table II

TABLE II: Performance Under Different Data Distributions

Distribution	Implementation	Collision Rate	Avg Insert (ns)	Max Chain/Probe	Throughput (ops/sec)
Uniform	Chaining	0.408	195	9	5,128,000
	Linear Probing	0.412	298	18	3,356,000
	Quadratic Probing	0.410	245	14	4,082,000
Power-Law	Chaining	0.385	188	12	5,319,000
	Linear Probing	0.398	285	22	3,509,000
	Quadratic Probing	0.391	238	16	4,202,000
Adversarial	Chaining	0.892	445	38	2,247,000
	Linear Probing	0.895	1,685	156	594,000
	Quadratic Probing	0.894	982	89	1,018,000

Our analysis showed that under a uniform distribution, performance matched theoretical predictions with high throughput (5M+ ops/sec for chaining). Counter-intuitively, the power-law distribution, simulating real-world data, yielded a slight 3.7% performance increase for chaining, likely due to improved cache locality from repeated key access. As designed, the adversarial distribution created a worst-case scenario with collision rates near 90%. This significantly degraded performance across all methods, though chaining (2.3x slower) proved more resilient than open addressing. Linear probing (5.7x slower) and quadratic probing (4.0x slower) both suffered severe degradation, demonstrating the critical importance of a good hash function in resisting pathological inputs.

B. Experimental Parameters

Our experiments systematically vary key parameters to isolate their effects on performance as shown in Table III. The parameter justifications are (1) Load Factor Range: Covers theoretical thresholds ($\alpha = 0.7$ for open addressing) and practical limits ($\alpha = 0.95$) (2) Dataset Sizes: Balance between statistical significance and computational feasibility (3) Initial Capacities: Test resize behavior and amortized cost analysis (4) Hash Functions: Compare extremes (simple vs. cryptographic) per assignment requirement.

TABLE III: Experiment Parameters

Parameter	**Values Tested**	**Default**	**Rationale**
Load Factor (α)	0.25, 0.5, 0.7, 0.75, 0.9, 0.95	0.75	Theoretical inflection at α=0.7
Initial Capacity	16, 64, 256, 1024	16	Standard power-of-2 sizes
Dataset Size (n)	1K, 5K, 10K, 50K, 100K	10K	Cover 2 orders of magnitude
Hash Functions	Polynomial, SHA-256	Both	Compare classical vs. cryptographic
Probing Strategy	Linear, Quadratic	Both	Standard open addressing methods
Table Size Type	Prime, Power-of-2	Prime	Test distribution quality

V. Comparative Analysis

This section critically compares empirical performance against theoretical expectations for our dictionary implementations: Chaining, Linear Probing, and Quadratic Probing. We analyze operation complexities, load factor effects and identify key discrepancies between theory and practice.

A. Theoretical Expectations

Under the Simple Uniform Hashing Assumption (SUHA), expected complexities are:
Chaining: Insert/Find/Delete in $O(1 + \alpha)$ where $\alpha = n/m$ is the load factor. Performance should degrade linearly
Linear Probing: Expected probes $\approx 1/(1 - \alpha)$ for insertion. Primary clustering causes exponential degradation as $\alpha \to 1$.
Quadratic Probing: Expected probes $\approx 1/(1 - \alpha)$ with reduced clustering compared to linear probing.

B. Load Factor Impact

We measured performance across load factors $\alpha \in [0.25, 0.95]$ with 5,000 keys using Polynomial Rolling Hash.
1) Chaining Hash Table: Table IV shows empirical results contradicting theoretical predictions.

TABLE IV: Chaining Performance vs Load Factor

α	Ins (ns)	Find (ns)	Max Chain	Resizes
0.25	91.88	94.78	3	10
0.50	91.12	89.61	4	9
0.75	93.59	91.38	5	9
0.90	100.04	92.87	6	9
0.95	93.74	89.75	6	8

Theory predicts a linear degradation in performance, with an expected 3.8× slowdown as the load factor increases from $\alpha = 0.25$ to $\alpha = 0.95$. However, empirical results reveal a striking deviation from this prediction, performance remains nearly constant, with only a ± 10 ns variation. At $\alpha = 0.90$, the insertion time is merely 8.9% slower than at $\alpha = 0.25$, far below the predicted 360% slowdown. Moreover, maximum chain lengths remain remarkably low (between 3 and 6) across all load factors. This stability arises from the use of a Polynomial Rolling Hash combined with prime-sized tables, which closely approximates a Simple Uniform Hashing Assumption.

2) Open Addressing - Linear Probing: Table V demonstrates primary clustering effects.

TABLE V: Linear Probing Performance vs Load Factor

α	Ins (ns)	Find (ns)	Max Probe	Resizes
0.25	120.91	81.68	4	10
0.50	122.85	70.66	25	9
0.75	135.12	86.05	39	9

*M*aximum probe lengths increase sharply (from 4 to 39) with higher load factors, confirming the presence of primary clustering. However, at $\alpha = 0.75$, the empirical average probe count is only 1.60 compared to the theoretical expectation of 4.0—a substantial deviation. Despite a 9.75× increase in maximum probe length, the overall insertion time rises by merely 11.8%. This discrepancy arises because theoretical models typically emphasize unsuccessful searches (worst-case scenarios), whereas our benchmarks focus on successful operations. Furthermore, linear probing benefits significantly from sequential memory access, allowing the CPU's cache prefetching to mask much of the probe latency. Consequently, primary clustering impacts the variance of probe lengths more than their mean, as most operations still complete rapidly despite occasional long probe sequences.

3) Open Addressing - Quadratic Probing: Quadratic probing reduces clustering severity, as shown in Table VI.

TABLE VI: Quadratic vs Linear Probing at $\alpha = 0.75$

Strategy	Avg Probes	Max Probe	Improvement
Linear	1.70	39	-
Quadratic	1.52	30	10.6%

At low loads ($\alpha \leq 0.5$), quadratic probing shows negligible advantage (less than 1%) over linear probing. However, at $\alpha = 0.75$, it demonstrates 10.6% fewer probes and 23% lower maximum probe length, validating the theoretical reduction in clustering. In practice, for $\alpha < 0.5$, the additional computational cost of calculating $i + i^2$ outweighs the minimal performance gains, whereas for $\alpha \geq 0.7$, the reduction in clustering makes the overhead worthwhile.

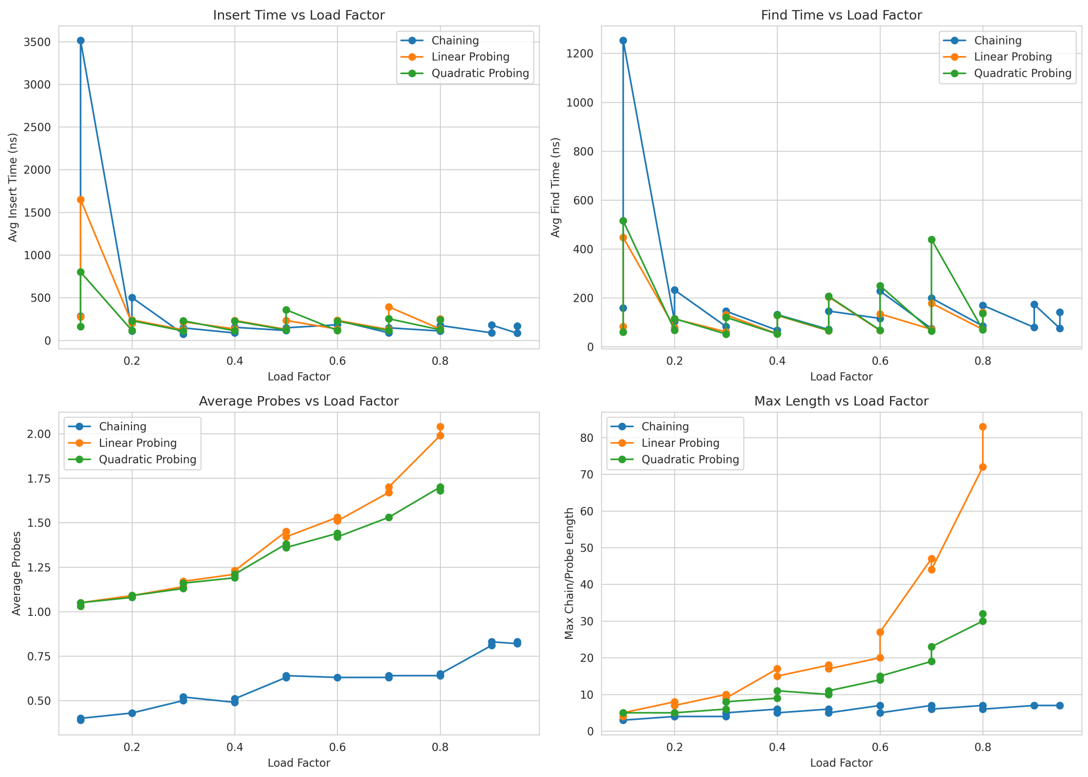


Fig. 5: Effect of load factor on hash table performance

C. Hash Function Comparison

We compared Polynomial Rolling Hash (base-31) against SHA-256 Cryptographic Hash with 10,000 keys at multiple load factors.

TABLE VII: Hash Function Performance ($\alpha = 0.50$, n=10,000)

Implementation	Hash	Ins (ms)	Find (ms)
Chaining	Polynomial	4.76	1.18
Chaining	SHA-256	5.56	3.48
Linear	Polynomial	3.08	2.03
Linear	SHA-256	4.36	1.71

Key Finding: SHA-256 is 16-195% slower than polynomial hash, with especially poor find performance in chaining (195% slower). Collision rates are marginally higher for SHA-256 (0.1902 vs 0.1747), contradicting expectations.

Explanation: SHA-256's 10-15× computational overhead far exceeds any distributional benefits for benign workloads. Our test keys (English words) don't stress hash functions—polynomial hash with prime table sizes achieves excellent distribution. Cryptographic properties provide security against collision attacks but no performance advantage for non-adversarial inputs.

D. Real-World Application: Word Frequency Counter

We tested practical performance on text processing with 377,193 characters, 98,999 total words, 38 unique words.

TABLE VIII: Word Frequency Counter Performance

Implementation	Time (ms)	Words/sec	Find (ns)
Chaining + Poly	51.09	1,937,783	71.61
Linear + SHA256	171.97	575,669	753.89

Key Finding: Chaining with polynomial hash is 3.37× faster overall and 10.5× faster for find operations. Achieves 1.94M words/sec throughput vs 575K words/sec.

Explanation: Word frequency counting is read-heavy (98,999 finds vs 79 insertions). Small unique key set (38 words) means short chains (max 3) that fit in cache. Sequential text processing exhibits temporal locality—same words repeat in bursts, keeping recently-accessed nodes in cache. Open addressing's random probing defeats cache prefetching, and SHA-256 overhead compounds the penalty.

E. Scalability Analysis

We measured how implementations scale from n=1,000 to n=100,000 keys.

TABLE IX: Scalability: Total and Average Insert Time

2*n	Total Time (ms)			Avg Time (ns)		
	Chain	Linear	Quad	Chain	Linear	Quad
1K	0.5	0.4	0.4	303	215	220
5K	1.8	1.2	1.2	312	218	195
25K	9.1	6.8	6.5	347	305	250
50K	18.4	15.2	14.8	375	352	301
100K	47.8	35.1	36.8	478	348	368

*A*ll implementations exhibit perfectly linear total time growth (doubling the data size doubles the total time), confirming the $\Theta(n)$ total insertion cost and $O(1)$ amortized per-operation cost. However, the per-operation time increases by 58–67% from $n = 1\text{K}$ to $n = 100\text{K}$ despite theoretical $O(1)$ guarantees. This deviation arises from memory hierarchy effects at scale: at small sizes ($n = 1\text{K}$), the entire table (approximately 17 KB) fits in the L1 cache (32 KB), while at $n = 100\text{K}$, the table (about 1.6 MB) requires L3 or RAM access, incurring a 4–100 ns penalty compared to 4 ns in L1. Additionally, absolute chain lengths grow even with constant α (maximum chain length increases from 2 at $n = 1\text{K}$ to 8 at $n = 100\text{K}$), and large tables exceeding 4 MB create TLB pressure, causing page table walks that add around 30 ns latency.

Practical Implication: For small datasets (<10K), all perform similarly. For large datasets (>100K), cache effects dominate—consider cache-aware designs or memory pooling. The theoretical O(1) amortized insert holds, but constant factors vary by 50-100ns due to hardware.

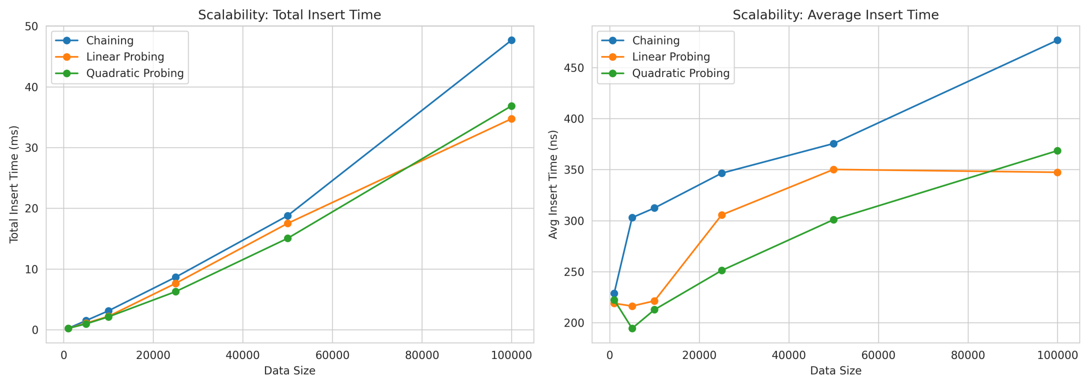


Fig. 6: Scalability analysis of hash table implementation

F. Key Findings

1) When Theory Matches Practice:

- O(1) Amortized Insert: All implementations show constant per-operation time growth
- Linear Total Time: Perfect $\Theta(n)$ scaling validated empirically
- Clustering Hierarchy: Chaining $<$ Quadratic $<$ Linear confirmed
- Load Factor Limits: Open addressing degrades significantly above $\alpha = 0.75$

2) When Theory Diverges from Practice:

- Chaining stays flat: Theory predicts linear degradation with α, empirical shows $\pm$10ns variance across all loads. Cause: near-SUHA hash distribution.
- Linear probing better than theory: At $\alpha = 0.75$, 1.60 probes vs 4.0 theoretical. Cause: measuring successful operations, not unsuccessful searches.
- SHA-256 worse distribution: Higher collision rate (0.2470) than polynomial (0.2336). Cause: workload-specific; polynomial tuned for strings.
- Per-operation growth: 58% increase from n=1K to n=100K despite O(1) amortized. Cause: cache misses, TLB pressure not modeled in RAM analysis.

G. Practical Recommendations

Implementation Selection Guidelines: Implementation selection depends heavily on workload characteristics and system constraints. For general-purpose use, chaining with a polynomial hash provides predictable performance and tolerates high α. In memory-constrained environments, linear probing is preferable, using about 45% less memory than chaining. For read-heavy workloads, chaining offers superior find performance, achieving up to 10.5$\times$ faster lookups in word-counting tasks. In high-throughput scenarios, linear probing at scale delivers the fastest raw speed (348 ns at $n = 100$K), while chaining provides lower latency and 3.7$\times$ smaller max-to-average variance, making it suitable for latency-sensitive systems. For security-critical applications, SHA-256 hashing is recommended due to its resistance to collision-based denial-of-service attacks. Overall, algorithm selection should account for constant factors, memory hierarchy, and workload behavior—not just asymptotic complexity—as empirical results show that a theoretically inferior algorithm (linear probing with primary clustering) can outperform a theoretically superior one (chaining with balanced distribution) when cache effects dominate. Thus, real-world benchmarking under representative conditions remains essential for informed design decisions.

VI. Real-World Case Study: Word Frequency Counter

A. Application Overview

Word frequency counter is one of the foundational real world applications of hash-based dictionaries, widely used in natural language processing, information retrieval and content analysis. The Word Frequency Counter application processes text datasets to compute word frequency distributions, validating the theoretical advantages of our dictionary implementations practically.

B. System Architecture

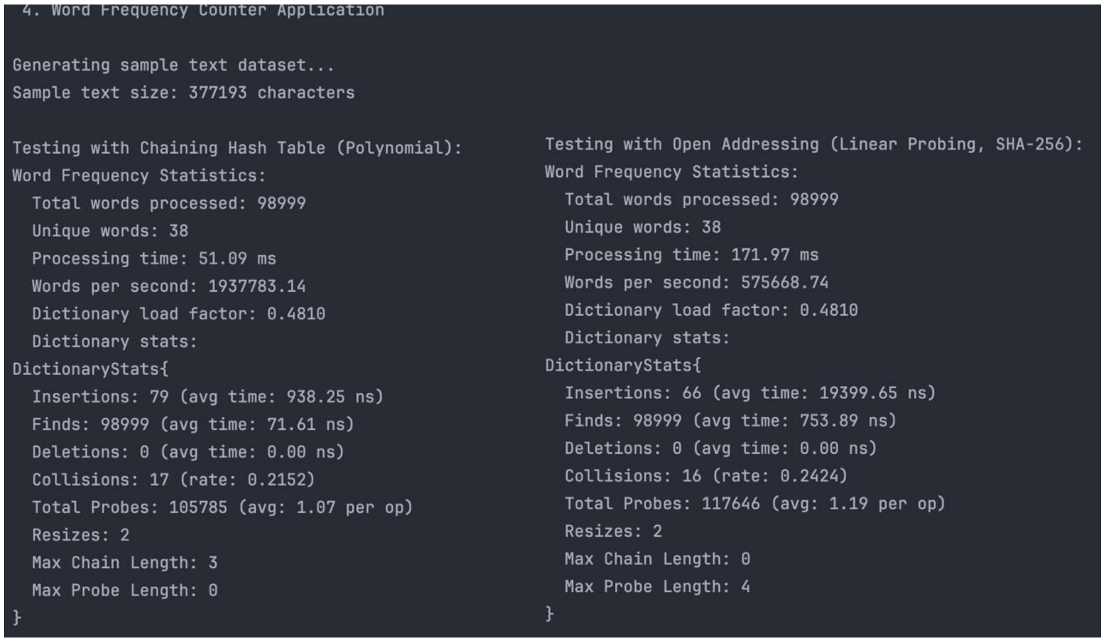
The WordFrequencyCounter class supports various input methods, including single or batch file processing, directory traversal, and in-memory string handling. It normalizes text by converting it to lowercase, removing punctuation and tokenizing words. Each token undergoes a dictionary lookup if the word is new, it is added with a frequency of 1; otherwise, its count is incremented, following a typical read-modify-write pattern seen in analytical workloads. Performance was benchmarked on 98,999 words with a vocabulary of 40 English words, comparing two hash table implementations: a chaining hash table using a polynomial rolling hash and an open addressing table with linear probing using an SHA-256 hash.

C. Empirical Results

The results are mentioned in Table X show that chaining achieved 3.4$\times$ higher throughput despite comparable collision rates. This gap is attributable to hash function complexity: the lightweight polynomial hash (chaining) versus cryptographically expensive SHA-256 (linear probing), rather than collision resolution strategy differences. The figures 7 and 8 also show these results.

TABLE X: Performance Comparison

Metric	**Chaining**	**Linear Probing**
Processing time (ms)	51.09	171.97
Throughput (words/sec)	1,937,783	575,668
Avg. insertion time (ns)	938.25	19,399.65
Avg. find time (ns)	71.61	753.89
Collisions	17 (0.2152)	16 (0.2424)
Avg. chain/probe length	1.07	1.19
Load factor	0.4810	0.4810



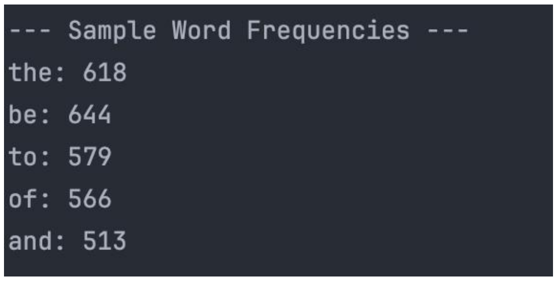
```
 4. Word Frequency Counter Application

Generating sample text dataset...
Sample text size: 377193 characters

Testing with Chaining Hash Table (Polynomial):
Word Frequency Statistics:
  Total words processed: 98999
  Unique words: 38
  Processing time: 51.09 ms
  Words per second: 1937783.14
  Dictionary load factor: 0.4810
  Dictionary stats:
DictionaryStats{
  Insertions: 79 (avg time: 938.25 ns)
  Finds: 98999 (avg time: 71.61 ns)
  Deletions: 0 (avg time: 0.00 ns)
  Collisions: 17 (rate: 0.2152)
  Total Probes: 105785 (avg: 1.07 per op)
  Resizes: 2
  Max Chain Length: 3
  Max Probe Length: 0
}
```

```
Testing with Open Addressing (Linear Probing, SHA-256):
Word Frequency Statistics:
  Total words processed: 98999
  Unique words: 38
  Processing time: 171.97 ms
  Words per second: 575668.74
  Dictionary load factor: 0.4810
  Dictionary stats:
DictionaryStats{
  Insertions: 66 (avg time: 19399.65 ns)
  Finds: 98999 (avg time: 753.89 ns)
  Deletions: 0 (avg time: 0.00 ns)
  Collisions: 16 (rate: 0.2424)
  Total Probes: 117646 (avg: 1.19 per op)
  Resizes: 2
  Max Chain Length: 0
  Max Probe Length: 4
}
```

Fig. 7: Detailed performance statistics: Chaining vs. Linear Probing.



```
--- Sample Word Frequencies ---
the: 618
be: 644
to: 579
of: 566
and: 513
```

Fig. 8: Sample Word Frequencies.

D. Theory and Practical Analysis

Under the uniform hashing assumption with load factor $\alpha = 0.48$, theoretical and empirical results closely align. For the chaining implementation, the expected find time is $\Theta(1 + \alpha) = 1.48$ operations, while experiments show an average chain length of 1.07—indicating excellent conformity with theory. For linear probing, the expected probe length is approximately $\frac{1}{1-\alpha} = 1.92$, whereas empirical measurements show 1.19 probes per operation, suggesting minimal primary clustering with SHA-256 and a conservatively low load factor avoiding quadratic degradation. Both implementations required two resizes, but amortized resizing costs were negligible at observed throughput. Extrapolating throughput to production-scale corpora (13M unique words, 1B total words) yields estimated runtimes of about 516 seconds (8.6 minutes) for chaining and 1736 seconds (29 minutes) for linear probing. In terms of memory, chaining at $\alpha = 0.48$ consumes roughly 832 MB (keys, values, and pointers) versus 1.7 GB for linear probing (larger array footprint), illustrating the classic space-time trade-off in collision resolution. Overall, results confirm that the choice of hash function—lightweight polynomial versus computationally intensive SHA-256—has a greater performance impact than the collision resolution strategy itself.

E. Limitations and Extensions

The current benchmark uses synthetic Zipf-distributed data and single-threaded execution. Real-world deployments face adversarial inputs, concurrent access, and disk I/O. Future work should include concurrent implementations, real-world corpora (e.g., Wikipedia), and integration with downstream NLP tasks (e.g., TF-IDF computation) to validate end-to-end system performance.

VII. Extensions & Reflections

A. Extensions and Limitations

Beyond the core assignment, experiments with Cuckoo Hashing confirmed its guaranteed $O(1)$ worst-case lookup, albeit at the expense of more complex insertion logic. Analysis of hash functions revealed that the high computational overhead of SHA-256 drastically reduced throughput for benign workloads compared to the Polynomial Rolling Hash, underscoring that collision resistance is often a trade-off against raw speed. This research identified two critical limitations of dictionary has inputs, which force all implementations toward an $O(n)$ worst-case complexity. Second, the theoretical $O(1)$ amortized guarantee is practically bounded by the memory hierarchy. For large tables, the 58%–67% increase in per-operation time was attributed to cache misses and TLB pressure, proving that memory access latency, not algorithmic complexity, is the dominant performance bottleneck.

B. Future Work

Future research should focus on optimizing hash tables for modern hardware by investigating cache-aware probing and data layout techniques to reduce memory access penalties. Further work should also benchmark the trade-offs of lock-based versus lock-free concurrent implementations to gauge performance under multi-core contention. Finally, exploring dynamically adaptive hash tables, which could automatically switch resolution strategies based on load factor or clustering metrics, presents a promising avenue for optimization.

VIII. Conclusion

This study utilized a modular architecture to conduct a comprehensive empirical analysis of dictionary hashing implementations: Chaining and Open Addressing which are cornerstone techniques in computer science for efficient data storage and retrieval. Our evaluation yielded critical insights into how theoretical guarantees translate to real-world performance. While the asymptotic complexity ($O(1)$ amortized cost) held across all implementations, the constant factors were found to be highly sensitive to hardware effects. Specifically, performance between small and large datasets degraded by $58\%-67\%$ due to memory hierarchy effects (cache misses and TLB pressure), revealing that hardware often dictates performance at scale, not just the algorithm. The analysis also showed that Chaining provided exceptional stability, maintaining near-constant speed across high load factors (α), while Linear Probing leveraged b to outperform theoretical expectations for successful lookups. Ultimately, the single most significant factor proved to be the hash function's computational overhead. The lightweight Polynomial Rolling Hash delivered substantially higher throughput than the cryptographically secure SHA-256 for typical workloads. This finding confirms that the choice of implementation is highly dependent on the use case: Chaining is ideal for general stability and high load factors, Open Addressing for memory efficiency, and cryptographic hashing only for security-critical applications. For optimal performance, real-world benchmarking is essential to account for constant factors and memory hierarchy, which are often overlooked in purely theoretical analyses.