CSE-208 Offline 3: Report

Mohammed Alim Uddin
Student ID: 2205175

July 4, 2025

1 Introduction

In this Offline assignment, we implemented and analyzed three distinct hashing techniques: **separate chaining**, **linear probing**, and **double hashing**.

We evaluated each method with varying load factors, ranging from $\alpha = 0.4$ to $\alpha = 0.9$, using randomly generated string data. The report analyzes their performance in terms of collision count, average search time, and probe count before and after deletions. All implementations use carefully selected hash functions and probing strategies to ensure fair and realistic comparisons.

2 Hash Functions

Two custom hash functions were used in this offlien assignment to map string keys to indices in the hash table. These functions were selected for their speed, simplicity, and good distribution properties.

2.1 Hash Function 1: Polynomial Rolling Hash

```cpp
int hash1(const string &key, int tableSize)
{
    long long hash_val = 0;
    const int prime = 31;
    long long prime_power = 1;

    for (char c : key)
    {
        hash_val = (hash_val + (c - 'a' + 1) * prime_power) % tableSize
            ;
        prime_power = (prime_power * prime) % tableSize;
    }
    return static_cast<int>(hash_val);
}
```

Listing 1: Hash Function 1: Polynomial Rolling Hash Implementation

This function implements the well-known polynomial rolling hash technique, where each character contributes to the hash value with increasing powers of a small prime number. The use of modular arithmetic with a prime table size ensures good pseudo-random distribution of keys.

2.2 Hash Function 2: djb2 Hash

```cpp
int hash2(const string &key, int tableSize)
{
    unsigned long long hash_val = 5381;

    for (char c : key)
    {
        hash_val = ((hash_val << 5) + hash_val) + c;
        hash_val %= tableSize;
    }
    return static_cast<int>(hash_val);
}
```

Listing 2: Hash Function 2: djb2 Hash Function Implementation

This is a variant of the djb2 hash function, developed by Dan Bernstein. It uses bit-shifting and accumulation to rapidly generate hash values from string data. Due to its simplicity and effectiveness, djb2 is widely adopted in hash table implementations. Its outputs are well-spread for typical string inputs, and it is particularly useful as a secondary hash in double hashing because it avoids producing zero values.

2.3 The reason behind choosing these hash functions : Efficiency and Distribution

Both functions operate in $\mathcal{O}(1)$ time for the input sizes used in this project, since average word length is bounded (typically 5–10 characters). Their use of prime constants and arithmetic ensures uniform key distribution across the hash table, reducing clustering and improving overall hash table performance. A good hash function is supposed to maintain uniformity and also time efficiency, which both my two hash function maintain.

3 Constants Used

Several constants were selected to optimize the behavior of the hash tables:

- **Step size (S) in linear probing:** Set to 3, a small prime number, to ensure all slots in the table can eventually be probed and to avoid repeating probe sequences.
- **Hash table size (N):** Set to 11003, a prime number. Using a prime table size improves key distribution and ensures that double hashing can fully explore the table without cycles or missed indices. I tried to make the table as large as possible for testing out every possible case, thus taking this table size.

These constants were chosen based on empirical practice and theoretical guidance to ensure correctness and fair benchmarking across all hashing strategies.

4 Uniqueness Test of Hash Functions

In order to ensure the effectiveness of a hash function, it is important to verify that it distributes keys uniformly across the table. One way to do this is to measure the **percentage of unique hash values** generated when a set of keys is inserted.

A good hash function should ideally avoid clustering and collisions, even when the load factor is high. According to our empirical study, both hash functions used in this assignment (Polynomial Rolling Hash and djb2) consistently generate a high percentage of unique hashes—well above the minimum threshold of 60% mentioned in the assignment, even at load factor $\alpha = 0.9$.

The following table shows the observed percentage of unique hash values for different load factors α, using a table size of 11003 and 10000 randomly generated strings.

Table 1: Percentage of Unique Hash Values for Varying Load Factors

Load Factor (α)	**Hash1 (Polynomial Rolling)**	**Hash2 (djb2)**
0.4	82.87%	82.10%
0.5	78.66%	79.19%
0.6	75.22%	75.62%
0.7	71.93%	72.37%
0.8	68.98%	69.29%
0.9	65.90%	66.55%

As seen above, both hash functions perform well across all tested load factors. While a slight drop in uniqueness is expected at higher α values due to increased table occupancy, the uniqueness remains significantly above the threshold, confirming the effectiveness of both hash strategies for this assignment.

5 Performance Tables

Each table on the following pages summarizes the performance metrics for a specific load factor, restructured to fit vertically within the page. We compare each method (chaining, linear probing, and double hashing) under both hash functions.

Table 2: Performance Metrics at Load Factor $\alpha = 0.4$

Metric	**Hash1 (Polynomial Rolling)**	**Hash2 (djb2)**
Separate Chaining with Balanced BST		
Insert Collisions	787	794
Avg Search Time (Before)	866.17 ns	702.79 ns
Avg Probes (Before)	N/A	N/A
Avg Search Time (After)	1150.00 ns	2454.55 ns
Avg Probes (After)	N/A	N/A
Linear Probing with Step Adjustment		
Insert Collisions	1549	1532
Avg Search Time (Before)	715.06 ns	523.74 ns
Avg Probes (Before)	1.35	1.35
Avg Search Time (After)	1152.27 ns	2456.82 ns
Avg Probes (After)	2.46	2.54
Double Hashing (Hash1 primary, Hash2 secondary)		
Insert Collisions	1230	
Avg Search Time (Before)	1309.48 ns	
Avg Probes (Before)	1.28	
Avg Search Time (After)	1793.18 ns	
Avg Probes (After)	2.09	
Double Hashing (Hash2 primary, Hash1 secondary)		
Insert Collisions	1195	
Avg Search Time (Before)	1630.77 ns	
Avg Probes (Before)	1.27	
Avg Search Time (After)	1834.09 ns	
Avg Probes (After)	2.08	

Table 3: Performance Metrics at Load Factor $\alpha = 0.5$

Metric	**Hash1 (Polynomial Rolling)**	**Hash2 (djb2)**
Separate Chaining with Balanced BST		
Insert Collisions	1202	1169
Avg Search Time (Before)	830.94 ns	733.14 ns
Avg Probes (Before)	N/A	N/A
Avg Search Time (After)	1370.91 ns	0.00 ns
Avg Probes (After)	N/A	N/A
Linear Probing with Step Adjustment		
Insert Collisions	2825	2788
Avg Search Time (Before)	602.07 ns	490.46 ns
Avg Probes (Before)	1.51	1.51
Avg Search Time (After)	0.00 ns	0.00 ns
Avg Probes (After)	2.94	3.04
Double Hashing (Hash1 primary, Hash2 secondary)		
Insert Collisions	2189	
Avg Search Time (Before)	864.21 ns	
Avg Probes (Before)	1.40	
Avg Search Time (After)	990.91 ns	
Avg Probes (After)	2.35	
Double Hashing (Hash2 primary, Hash1 secondary)		
Insert Collisions	2084	
Avg Search Time (Before)	803.13 ns	
Avg Probes (Before)	1.38	
Avg Search Time (After)	0.00 ns	
Avg Probes (After)	2.35	

Table 4: Performance Metrics at Load Factor $\alpha = 0.6$

Metric	**Hash1 (Polynomial Rolling)**	**Hash2 (djb2)**
Separate Chaining with Balanced BST		
Insert Collisions	1636	1637
Avg Search Time (Before)	639.30 ns	529.92 ns
Avg Probes (Before)	N/A	N/A
Avg Search Time (After)	0.00 ns	1043.94 ns
Avg Probes (After)	N/A	N/A
Linear Probing with Step Adjustment		
Insert Collisions	5281	4934
Avg Search Time (Before)	531.28 ns	472.35 ns
Avg Probes (Before)	1.80	1.75
Avg Search Time (After)	0.00 ns	0.00 ns
Avg Probes (After)	4.20	3.62
Double Hashing (Hash1 primary, Hash2 secondary)		
Insert Collisions	3503	
Avg Search Time (Before)	690.35 ns	
Avg Probes (Before)	1.53	
Avg Search Time (After)	1027.27 ns	
Avg Probes (After)	2.73	
Double Hashing (Hash2 primary, Hash1 secondary)		
Insert Collisions	3475	
Avg Search Time (Before)	644.75 ns	
Avg Probes (Before)	1.53	
Avg Search Time (After)	1056.06 ns	
Avg Probes (After)	2.79	

Table 5: Performance Metrics at Load Factor $\alpha = 0.7$

Metric	**Hash1 (Polynomial Rolling)**	**Hash2 (djb2)**
Separate Chaining with Balanced BST		
Insert Collisions	2172	2152
Avg Search Time (Before)	847.70 ns	762.14 ns
Avg Probes (Before)	N/A	N/A
Avg Search Time (After)	716.88 ns	870.13 ns
Avg Probes (After)	N/A	N/A
Linear Probing with Step Adjustment		
Insert Collisions	9908	8398
Avg Search Time (Before)	659.31 ns	475.20 ns
Avg Probes (Before)	2.29	2.09
Avg Search Time (After)	0.00 ns	876.62 ns
Avg Probes (After)	6.81	5.46
Double Hashing (Hash1 primary, Hash2 secondary)		
Insert Collisions	5630	
Avg Search Time (Before)	846.53 ns	
Avg Probes (Before)	1.73	
Avg Search Time (After)	719.48 ns	
Avg Probes (After)	3.41	
Double Hashing (Hash2 primary, Hash1 secondary)		
Insert Collisions	5493	
Avg Search Time (Before)	720.59 ns	
Avg Probes (Before)	1.71	
Avg Search Time (After)	719.48 ns	
Avg Probes (After)	3.47	

Table 6: Performance Metrics at Load Factor $\alpha = 0.8$

Metric	**Hash1 (Polynomial Rolling)**	**Hash2 (djb2)**
Separate Chaining with Balanced BST		
Insert Collisions	2742	2722
Avg Search Time (Before)	703.82 ns	486.37 ns
Avg Probes (Before)	N/A	N/A
Avg Search Time (After)	710.23 ns	0.00 ns
Avg Probes (After)	N/A	N/A
Linear Probing with Step Adjustment		
Insert Collisions	19220	19174
Avg Search Time (Before)	551.01 ns	391.16 ns
Avg Probes (Before)	3.18	3.18
Avg Search Time (After)	1604.55 ns	730.68 ns
Avg Probes (After)	11.90	12.93
Double Hashing (Hash1 primary, Hash2 secondary)		
Insert Collisions	8943	
Avg Search Time (Before)	699.61 ns	
Avg Probes (Before)	2.02	
Avg Search Time (After)	777.27 ns	
Avg Probes (After)	4.59	
Double Hashing (Hash2 primary, Hash1 secondary)		
Insert Collisions	8831	
Avg Search Time (Before)	703.82 ns	
Avg Probes (Before)	2.00	
Avg Search Time (After)	775.00 ns	
Avg Probes (After)	4.56	

Table 7: Performance Metrics at Load Factor $\alpha = 0.9$

Metric	Hash1 (Polynomial Rolling)	Hash2 (djb2)
Separate Chaining with Balanced BST		
Insert Collisions	3393	3343
Avg Search Time (Before)	677.64 ns	529.29 ns
Avg Probes (Before)	N/A	N/A
Avg Search Time (After)	691.92 ns	667.68 ns
Avg Probes (After)	N/A	N/A
Linear Probing with Step Adjustment		
Insert Collisions	49723	49160
Avg Search Time (Before)	657.75 ns	520.00 ns
Avg Probes (Before)	6.02	5.96
Avg Search Time (After)	1390.91 ns	1378.79 ns
Avg Probes (After)	36.64	31.36
Double Hashing (Hash1 primary, Hash2 secondary)		
Insert Collisions	15587	
Avg Search Time (Before)	736.82 ns	
Avg Probes (Before)	2.57	
Avg Search Time (After)	695.96 ns	
Avg Probes (After)	7.57	
Double Hashing (Hash2 primary, Hash1 secondary)		
Insert Collisions	15425	
Avg Search Time (Before)	785.90 ns	
Avg Probes (Before)	2.56	
Avg Search Time (After)	1402.02 ns	
Avg Probes (After)	7.69	

6 Analysis and Discussion

This section analyzes the performance of separate chaining, linear probing, and double hashing, drawing insights from the empirical data presented in the tables and correlating them with theoretical expectations. The **load factor** (α = number of elements/table size) is a critical metric, indicating how full a hash table is and directly impacting collision rates and search efficiency.

6.1 Separate Chaining using Red-Black Tree

Theoretically, the time complexity for successful and unsuccessful searches in separate chaining with a balanced data structure like a Red-Black Tree (RBT) is $O(1+\alpha)$. This means performance should degrade gracefully as the load factor increases, as the '1' represents the initial hash computation and α relates to the average length of the chain (which is a balanced BST in this case).

Our data generally supports this observation:

- **Insert Collisions:** Separate chaining consistently shows the lowest number of insert collisions across all load factors. This is because "collision" in chaining simply means another element needs to be added to the BST at that hash index, rather than finding an alternative slot in the main table.
- **Average Search Time (Before Deletion):** The search times remain relatively low and stable, showing a gradual increase as α rises. The data table reflects the $O(1+\alpha)$ behavior, where the cost scales linearly with the average chain length.
- **Impact of Load Factor:** The performance of separate chaining is robust to increasing load factors. While collisions increase, the average search and insertion times do not degrade as drastically as open addressing methods, confirming its theoretical advantage for higher load factors. The "N/A" for probes is expected, as probing is a concept specific to open addressing.

6.2 Open Addressing using Linear Probing

Linear probing attempts to find the next available slot by incrementing the index by a fixed step (here, $S = 3$). This method is known to suffer from **primary clustering**, where long sequences of occupied slots form, increasing the number of probes significantly, especially at higher load factors. Its theoretical time complexity for operations can approach $O(\frac{1}{1-\alpha})$ for unsuccessful searches and insertions.

Our data strongly illustrates primary clustering effects:

- **Insert Collisions:** Linear probing exhibits the highest number of insert collisions. At $\alpha = 0.4$, collisions are already around 1500, skyrocketing to nearly 50,000 at $\alpha = 0.9$. This exponential increase in collisions with load factor is a hallmark of primary clustering.
- **Average Probes (Before Deletion):** The average probes dramatically increase as α approaches 1. From 1.35 at $\alpha = 0.4$ to over 6 at $\alpha = 0.9$. This confirms that the system has to "probe" much further to find an empty slot or the target key.
- **Average Probes (After Deletion):** The average probes after deletion are significantly higher than before deletion (e.g., 36.64 vs 6.02 at $\alpha = 0.9$). This is due to "tombstones" (marked deleted slots) which must still be traversed during search, effectively lengthening probe sequences and indicating **secondary clustering**.

- **Impact of Load Factor:** Linear probing shows the most severe performance degradation with increasing load factor. Its reliance on contiguous empty slots makes it highly sensitive to

- **Impact of Load Factor:** Linear probing shows the most severe performance degradation with increasing load factor. Its reliance on contiguous empty slots makes it highly sensitive to load. As α increases, not only do insertions take longer due to more collisions, but searches are also penalized heavily due to clustering and deleted markers.

6.3 Double Hashing

Double hashing uses a second hash function to compute the probe step, reducing both primary and secondary clustering. Theoretical analysis suggests that the average number of probes is better than linear probing, especially for unsuccessful searches, as the distribution of probe sequences is more uniform.

- **Insert Collisions:** Double hashing consistently outperforms linear probing in terms of insert collisions. At $\alpha = 0.9$, collisions remain within 15,000 compared to linear probing's 49,000+, showing that the double hash function effectively spreads out entries even in dense tables.
- **Average Probes:** Probe counts for double hashing scale reasonably with load factor. At $\alpha = 0.4$, it's around 1.28, and only rises to about 2.57 by $\alpha = 0.9$. This validates the theoretical claim that double hashing maintains a more uniform probe distribution and delays clustering.
- **Search Times:** The average search time trends match the probe counts, remaining better than linear probing, especially after deletion. At high load factors, while performance still degrades (as expected), it does so more gracefully.
- **Impact of Load Factor:** Double hashing is a better open addressing strategy for moderate to high load factors. Its reduced clustering makes it ideal for dynamic hash tables where maintaining probe efficiency is critical.

7 Conclusion

Our experimental results support the theoretical behavior of different hashing techniques. We summarize our conclusions below:

- **Separate chaining** offers consistent performance across all load factors. It is most suitable when the hash table size cannot be tightly managed or when deletions are frequent.
- **Linear probing** performs well only for low load factors ($\alpha \leq 0.6$). Its simplicity and good cache performance make it attractive in memory-constrained environments, but only under light loads.

- **Double hashing** strikes the best balance between speed, collision avoidance, and memory efficiency. It is more robust than linear probing as α increases.