Hashing with Worst Case Constant Search Time

Mini Project Report

Group Members:

Tanuj Agarwal	241110076
Harsh Baid	241110026
Ayush	220259

Group ID: 16

Department of Computer Science
Indian Institute of Technology Kanpur
2024-2025

Contents

1 Introduction

Hashing is a widely used technique that allows storing and searching for elements in constant time on average. However, standard hashing techniques may suffer from collisions, leading to worst-case search times greater than $O(1)$. The goal of this project is to implement a hashing technique that guarantees worst-case $O(1)$ search time, making it highly efficient compared to other search methods such as binary search on a sorted array.

1.1 Problem Definition

Given a universe of elements $U = \{1, 2, \ldots, m\}$, a subset $S \subseteq U$ of size s needs to be stored in a data structure that supports fast search queries. The challenge is that $s \ll m$, meaning that the subset S is significantly smaller than the universe. For instance, in real-world scenarios, m could be as large as 10^{18}, while s might only be 10^3. The objective is to construct a data structure that efficiently answers search queries of the form:

"Does element i belong to S?"

for any $i \in U$, ensuring that the worst-case search time remains constant. This project explores a static hashing approach that achieves this performance and compares it with alternative methods like binary search on sorted arrays.

1.2 Simple hash function ($i \mod n$)

A simple hash function of the form:

$$h(i) = i \mod n$$

where $i \in U$ and n is the size the table, is observed to work well in practice. This is due to the fact the the set S is usually a uniformly random subset of U.

Proof: Let $y_1, y_2, \ldots, y_s$ denote s elements selected randomly uniformly from $\mathcal{U}$ to form $\mathcal{S}$.
Number of elements colliding with y_1 is $\lfloor \frac{m}{n} \rfloor - 1$

Therefore, $\mathbb{P}(y_j \text{ collides with } y_1) =$

$$\frac{\lfloor \frac{m}{n} \rfloor - 1}{m-1} \leq \frac{1}{n}$$

Expected number of collisions is

$$= \frac{1}{n}(s-1) < \frac{s}{n}$$

$$= O(1) \quad \text{for } n = O(s)$$

Therefore, the probability that a given element y_i collides with another element y_j in the hash function is:

$$P(y_j \text{ collides with } y_i) = \frac{1}{n}$$

.

Therefore, we can say that this simple hash function works well due to the reason that S is a uniformly random subset of U .

2 Universal Hashing

If a malicious adversary chooses the keys to be hashed by some fixed hash function like the one shown above, then the adversary can choose n keys that will all hash to the same slot, yielding an average retrieval time of $O(n)$. Any fixed hash function is vulnerable to such terrible worst-case behavior; the only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored. This approach is called ***universal hashing*** and it can yield provably good performance on average, no matter which keys the adversary chooses.

Definition: A collection $\mathcal{H}$ of hash-functions is said to be c-universal if there exists a constant c such that for any $i, j \in \mathcal{U}$,

$$\mathbb{P}_{h \in_r \mathcal{H}}\left(h(i) = h(j)\right) \leq \frac{c}{n}$$

2.1 2-Universal hash function

A collection $\mathcal{H}$ of hash function is said to be **2-universal** if for any $i, j \in \mathcal{U}$,

$$\mathbb{P}_{h \in_r \mathcal{H}}\left(h(i) = h(j)\right) \leq \frac{2}{n}$$

Consider the function defined as:

$$f_x(i) = ix \bmod p$$

$$h_x(i) = f_x(i) \bmod n$$

Here, p is a prime number, $x \in \{1, 2, \ldots, p-1\}$. The family of hash functions is given by:

$$\mathcal{H} = \{h_x | x \in [1, p-1]\}$$

Theorem: $\mathcal{H}$ is a 2-universal family of hash functions.

Proof: Let p be a prime number and assume it is large enough so that every possible key is in the range 0 to $p-1$, inclusive. Let $\mathbb{Z}_p^*$ denote the set $\{1, 2, \ldots, p-1\}$. Consider two distinct keys k and l from {1, 2, . . . , p - 1} so that $k \neq l$. For a given hash function h_x where $x \in \mathbb{Z}_p^*$, define:

$$r = (xk) \bmod p, \quad s = (xl) \bmod p$$

It is evident that $r \neq s$. Since both x and $(k-l)$ are nonzero modulo p, their product is also nonzero modulo p. Now consider the equation:

$$(xk \bmod p) \bmod n = (xl \bmod p) \bmod n$$

$$x(k-l) \bmod p \bmod n = 0$$

$$x(k-l) \bmod p \in \{n, 2n, 3n, , p-n, p-2n, p-3n, \ldots\}$$

Since $k-l$ has a multiplicative inverse modulo p, the number of such x satisfying equation above is at most:

$$\leq \frac{2(p-1)}{n}$$

Therefore,

$$\Pr[x(k-l) \bmod p \bmod n = 0]$$

is equivalent to the number of satisfying x divided by the total number of values x can take. Hence,

$$\Pr[h_x(k) = h_x(l)] \leq \frac{2(p-1)/n}{p-1} \leq \frac{2}{n}$$

This shows that $\mathcal{H}$ is 2-universal hash function.

2.2 1-Universal hash function

A collection $\mathcal{H}$ of hash function is said to be **1-universal** if for any $i, j \in \mathcal{U}$,

$$\mathbb{P}_{h \in_r \mathcal{H}}\left(h(i) = h(j)\right) \leq \frac{1}{n}$$

Consider the function defined as:

$$f_{x,y}(i) = (ix + y) \bmod p$$

$$h_{x,y}(i) = f_{x,y}(i) \bmod n$$

Here, p is a prime number, $x \in \{1, 2, \ldots, p-1\}$, and $y \in \{0, 1, \ldots, p-1\}$. The family of hash functions is given by:

$$\mathcal{H} = \{h_{x,y} \,|\, x \in [1, p-1],\ y \in [0, p-1]\}$$

Theorem: $\mathcal{H}$ is a 1-universal family of hash functions.

Proof: Let p be a prime number and assume it is large enough so that every possible key is in the range 0 to $p-1$, inclusive. Let $\mathbb{Z}_p$ denote the set $\{0, 1, 2, \ldots, p-1\}$, and let $\mathbb{Z}_p^*$ denote the set $\{1, 2, \ldots, p-1\}$. Consider two distinct keys k and l from $\mathbb{Z}_p$, so that $k \neq l$. For a given hash function $h_{x,y}$, define:

$$r = (xk + y) \bmod p, \quad s = (xl + y) \bmod p$$

We first show that $r \neq s$.

$$r - s \equiv x(k - l) \pmod p$$

Since p is prime and both x and $(k-l)$ are nonzero modulo p, their product is also nonzero modulo p. Therefore, $r \neq s$, meaning that distinct inputs k and l map to distinct values r and s modulo p. So there are no collisions at the "mod p" level.

Moreover, each of the $p(p-1)$ choices for the pair (x, y), with $x \neq 0$, yields a different resulting pair (r, s) with $r \neq s$, since we can solve for x and y given r and s:

$$x = \left[(r - s) \cdot (k - l)^{-1} \mod p\right]$$

$$y = (r - xk) \bmod p$$

where $(k-l)^{-1} \mod p$ denotes the unique multiplicative inverse of $(k-l) \mod p$. Since there are only $p(p-1)$ possible such distinct pairs (r, s), this establishes a one-to-one correspondence between the set of pairs (x, y) and the resulting values (r, s) with $r \neq s$.

Now, for any fixed pair of distinct inputs $k, l \in \mathbb{Z}_p$, if we pick (x, y) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting (r, s) pair is uniformly distributed among all $r \neq s \in \mathbb{Z}_p$.

The probability that $r \equiv s \pmod{n}$ (i.e., they collide after reduction modulo n) is at most the number of s values such that $s \neq r$ and $s \equiv r \mod n$, divided by the total number of possible s-values:

$$\text{Number of such } s \leq \left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p+n-1}{n} - 1 = \frac{p-1}{n}$$

Thus, the probability that $r \equiv s \pmod{n}$ is at most:

$$\frac{(p-1)/n}{p-1} = \frac{1}{n}$$

Hence, for any distinct $k, l \in \mathbb{Z}_p$,

$$\Pr[h_{x,y}(k) = h_{x,y}(l)] \leq \frac{1}{n}$$

3 Hashing with $O(1)$ Search time

3.1 Perfect Hashing using $O(s^2)$ space

Let $\mathbf{S}$ be any set of size s and U be the universe of elements.

Let $\mathcal{H}$ be a Universal Hash Family, such that

$$\mathbb{P}_{h \in_r \mathcal{H}}\left(h(i) = h(j)\right) \leq \frac{1}{n}$$

where $i, j \in U$

Let X: be a random variable denoting the number of collisions for $\mathbf{S}$ when $h \in_r \mathcal{H}$.

Let's calculate the $\mathbb{E}[X]$.

For each $i, j \in \mathbf{S}$, define:

$$X_{i,j} = \begin{cases} 1 & \text{if } h(i) = h(j) \\ 0 & \text{otherwise} \end{cases}$$

$$X = \sum_{\substack{i<j \\ i,j \in \mathbf{S}}} X_{i,j}$$

$$\mathbb{E}[X] = \sum_{\substack{i<j \\ i,j \in \mathbf{S}}} \mathbb{E}[X_{i,j}] = \sum_{\substack{i<j \\ i,j \in \mathbf{S}}} \mathbb{P}[X_{i,j} = 1] \leq \sum_{\substack{i<j \\ i,j \in \mathbf{S}}} \frac{1}{n}$$

$$\Rightarrow \mathbb{E}[X] \leq \frac{1}{n} \cdot \frac{s(s-1)}{2}$$

Now, fix $n = s^2$:

$$\mathbb{E}[X] \leq \frac{1}{s^2} \cdot \frac{s(s-1)}{2} \leq \frac{1}{2}$$

Hence, the expected number of collisions is:

$$\mathbb{E}[X] \leq \frac{1}{2}$$

Using Markov's Inequality:

$$\Pr(X \geq 1) \leq \frac{\mathbb{E}[X]}{1} = \frac{1}{2}$$

Therefore,

$$\Pr(\text{No collision}) = \Pr(X = 0) = 1 - \Pr(X \geq 1) \geq 1 - \frac{1}{2} = \frac{1}{2}$$

Hence, for $n = s^2$ there will be no collision with probability atleast $\frac{1}{2}$.

Algorithm

```
**Input:** A set $S \subseteq U$ of size $s$, and a universal hash family $\mathcal{H}$
**Output:** A perfect hash table $T$ such that search happens in $O(1)$ worst case time.
**repeat**
    Pick randomly a hash function $h \in_r \mathcal{H}$
    Set collision count $t \leftarrow 0$
    **for all** $i \in S$ **do**
        compute $h(i)$
        **if** $T[h(i)] \neq empty$ **then**
            $t = 1$; break;
        **end if**
    **end for**
**until** $t = 0$
**return** the perfect hash table $T$
```

Time complexity: The expected time taken by the algorithm is $O(s^2)$.
Space complexity: The expected space required by the algorithm is $O(s^2)$.
It performs search in $O(1)$ worst case time.

3.2 Hashing in $O(S)$ optimal space

Earlier we found the expected number of collisions as,

$$\mathbb{E}[X] \leq \frac{1}{n} \cdot \frac{s(s-1)}{2}$$

If we fix $n = s$ we get the expected number of collisions as

$$\mathbb{E}[X] \leq \frac{1}{s} \cdot \frac{s(s-1)}{2} \leq \frac{(s-1)}{2}$$

Algorithm

```
**Input:** A set $S \subseteq U$ of size $s$, and a universal hash family $\mathcal{H}$
**Output:** A hash table $T$ such that search happens in $O(1)$ worst case time.
Fix $n = cs$
**Repeat**
Pick randomly a hash function $h \in_r \mathcal{H}$
```

```
Set collision count $t \leftarrow 0$
for all $i \in S$ do
    compute $h(i)$
    t=t+length($T[h(i)]$)
    add $i$ to end of the list
end forUntil $t \leq s$
Primary hash table is complete;

For each $0 \leq i < n$
If size of list $T[i] > 1$
    1. Build a perfect hash table for list $T[i]$;
    2. Make $T[i]$ point to this hash table;
```

Space Required: Now lets show the space required by the algorithm.
Let Z_i : number of elements in $T[i]$
Extra Space required :

$$\sum_{i<n \text{ and } Z_i>1} Z_i^2$$

$$X = \sum_{i<n \text{ and } Z_i>1} \frac{Z_i(Z_i-1)}{2}$$

$$\Rightarrow \sum_{i<n \text{ and } Z_i>1} Z_i^2 = 2X + \sum_{i<n \text{ and } Z_i>1} Z_i$$

X is the number of collisions and let $X = s$
Therefore,

$$\sum_{i<n \text{ and } Z_i>1} Z_i^2 = 2X + \sum_{i<n \text{ and } Z_i>1} Z_i$$

$$\Rightarrow \sum_{i<n \text{ and } Z_i>1} Z_i^2 \leq 3s$$

Hence, the extra space required for the algorithm is $\leq 3s$.

Time complexity: The expected time taken by the algorithm is $O(s)$.
Space complexity: The expected space required by the algorithm is $O(s)$.
It performs search in $O(1)$ worst case time.

4 Techniques and Implementation

4.1 Implementation Overview

We implemented and tested multiple hashing techniques, focusing on achieving constant-time search performance. Our goal was to empirically validate the theoretical guarantees of worst-case $O(1)$ search time under different data distributions. Below are the key techniques we explored:

1. **Simple Hashing using $i \mod n$:**
   We started with the classic hashing function $h(i) = i \mod n$. This approach is

simple and works well when the input data is uniformly distributed. However, we observed a significant degradation in performance when the data followed certain patterns, such as clustered values or arithmetic progressions. These patterns led to frequent collisions, increasing the worst-case search time.

2. **Perfect Hashing using Random Hash Functions:**
   We implemented perfect hashing by selecting a random hash function from a universal family of the form $h_{x,y}(i) = ((ix + y) \mod p) \mod n$, where x and y are chosen uniformly at random and p is a large prime and $n = s^2$. We repeatedly sampled hash functions until a collision-free assignment was found. This technique guarantees $O(1)$ search time in the worst case, at the cost of higher space complexity ($O(s^2)$).

3. **Two-Level Perfect Hashing:**
   While this method ensures constant lookup time, the quadratic space requirement is often impractical. To address this, we turn to a more space-efficient strategy. To optimize space usage while retaining worst-case $O(1)$ performance, we implemented a two-level hashing scheme. The first level hashes all elements into a primary table of size $n = 2s$ using a random hash function from a universal family of the form $h_{x,y}(i) = ((ix+y) \mod p) \mod n$, where x and y are chosen uniformly at random and p is a large prime. Buckets containing multiple elements undergo a secondary perfect hashing process: for each such bucket, we construct a secondary hash table of size t^2 (where t is the number of elements in the bucket) and randomly sample a new hash function until a collision-free mapping is found. The following pseudocode illustrates the secondary hash table construction:

4. **Collision Handling and Rehashing Strategy:**
   For both perfect hashing approaches, we use rehashing as a fallback strategy. If a chosen hash function results in a collision, we discard the function and repeat the process with a newly sampled pair (x, y). In the two-level scheme, rehashing is localized to the affected bucket, significantly improving efficiency over global rehashing. This probabilistic retry mechanism ensures that with high probability, a collision-free configuration is found in a reasonable number of attempts.

4.2 Experimental Setup

Programming Language and Libraries:
The implementation was done in `Python`, owing to its simplicity and efficiency for prototyping. For visualizing the empirical results, the `matplotlib` library was used to generate plots and compare performance across different datasets and hashing strategies.

Dataset Generation:
We evaluated our implementation using datasets of varying sizes to analyze scalability and performance across different input scales. Each dataset was a subset S sampled from a large fixed universe of size $m = 10^{18}$, representing practical scenarios where the key space is vast but only sparsely populated. Elements in S were primarily sampled uniformly at random to reflect average-case behavior.

To test robustness under more structured or challenging conditions, we also included adversarial datasets. These were constructed using arithmetic progressions (AP), where elements are evenly spaced, simulating inputs that could lead to poor performance in naive hashing schemes.

5 Results and Analysis

5.1 Progression of Approaches

We began by implementing a simple hashing approach using the function $h(i) = i \mod n$. While this method performed well on uniformly random datasets, it failed under structured inputs such as arithmetic progressions, leading to severe clustering and degraded performance.

To overcome this, we implemented the perfect hashing strategy using a universal hash function and a hash table of size s^2. This approach successfully guaranteed $O(1)$ worst-case search time. However, it incurred significant space overhead, especially for large s, making it impractical in memory-constrained environments.

We then adopted a two-level hashing scheme. This technique applies a universal hash function at the first level to split the dataset into buckets. For each bucket with collisions, a secondary perfect hash table is constructed using a new universal function and $O(k^2)$ space (where k is the bucket size). This method achieved both worst-case $O(1)$ query time and space efficiency close to $O(s)$ on average.

5.2 Comparative Evaluation

To evaluate the effectiveness of our final approach, we compared it against traditional search methods like binary search (on sorted arrays) and binary search trees (BST). The comparison was done across two types of datasets:

- **Uniform Random Data:** Elements are sampled randomly from the universe. This represents average-case input.
- **Arithmetic Progressions:** Structured sequences (e.g., 2, 4, 6, ...) to test performance on predictable and non-random data.

5.3 Observations and Insights

- **Mod-based Hashing:** Fastest on uniform data, but fails catastrophically on structured data like AP distributions. This can be clearly observed by the figure 1 shown below. As the data size increases the more elements gets mapped to the same index in case of AP distributions which in turn increases the average time taken for a search query.

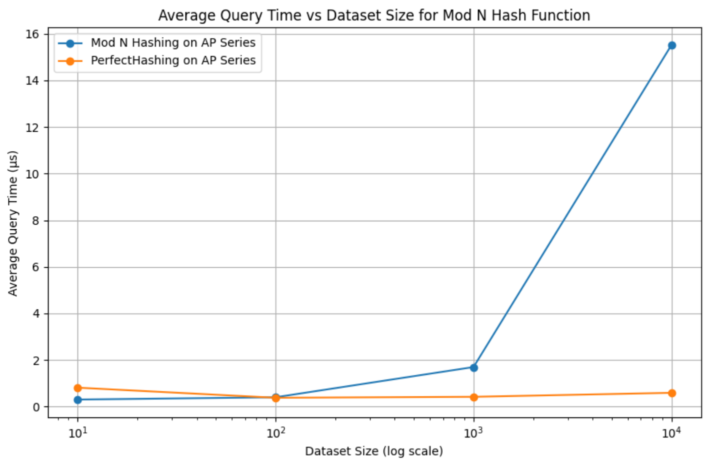

Figure 1: Simple hashing failing condition

- **Perfect Hashing (O(s^2)):** Guaranteed constant-time lookups but memory overhead became unsustainable beyond moderate s.

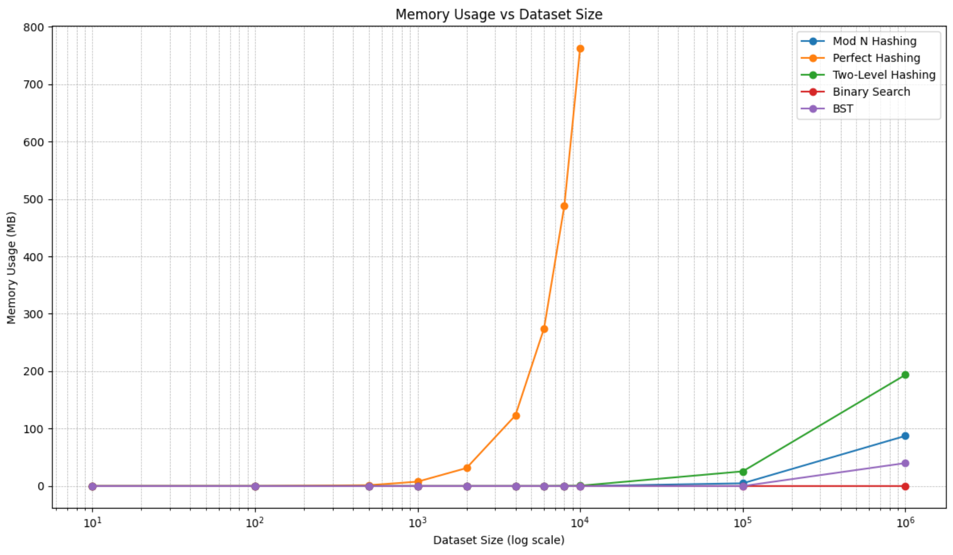

Figure 2: Memory usage for different algorithms

- **Two-Level Hashing:** Achieved the best of both worlds — fast lookups with high probability and space close to optimal.

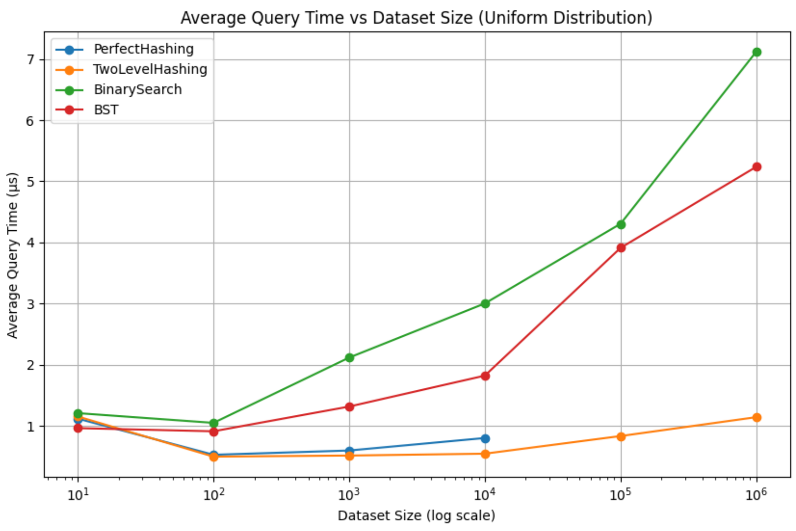

Figure 3: Comparison of average search times across various algorithms, sampled 1000 times

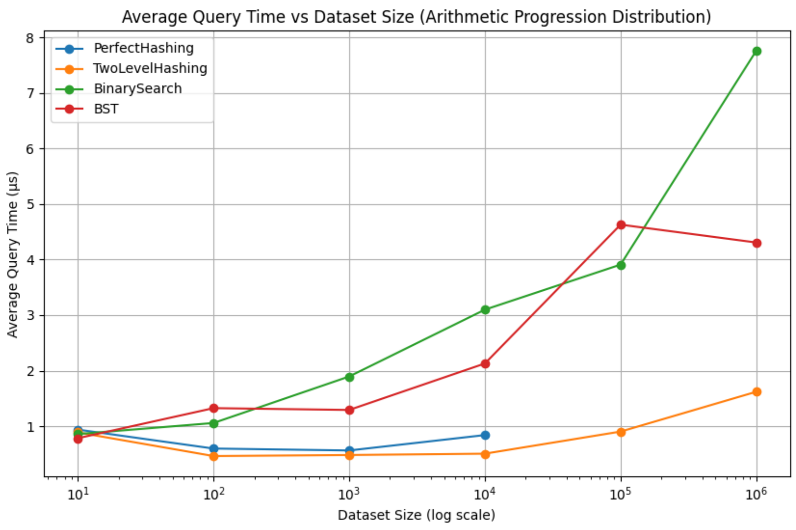

Figure 4: Comparison of average search times across various algorithms, sampled 1000 times

- **Binary Search and BST:** Performance grew with $\log s$; consistent but slower than hashing in large datasets. BSTs especially lagged behind due to pointer-based navigation overhead.

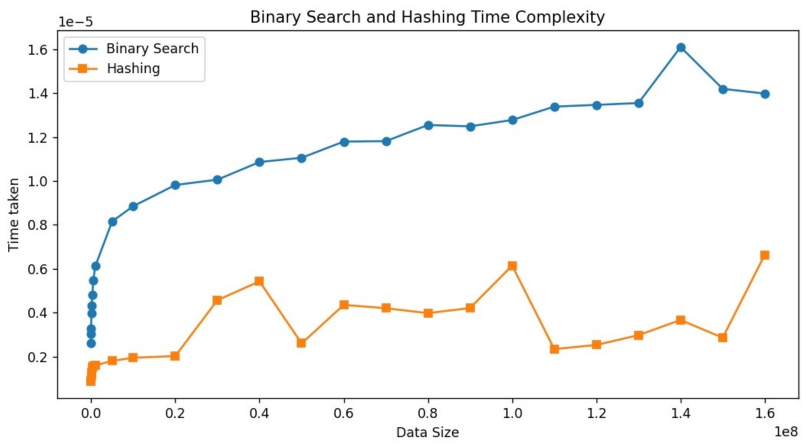

Figure 5: Query time comparison for two-level hashing and binary search on huge dataset

- **Empirical Results on Uniform Distribution**

  To better understand the practical performance of each approach, we conducted empirical tests to measure the average query time (in microseconds) across various dataset sizes. The table below summarizes the results for uniformly distributed datasets. As expected, both perfect and two-level hashing outperform comparison-based methods. However, perfect hashing becomes impractical for large datasets due to memory constraints, which is why results for sizes beyond 10^4 are omitted. Two-level hashing maintains consistent performance, validating its scalability and efficiency.

Dataset Size	Perfect Hashing	Two Level Hashing	Binary Search	Binary Search Tree
10	0.77	0.79	0.84	0.65
100	0.46	0.35	0.75	0.96
1000	0.41	0.35	1.33	0.90
10000	0.51	0.40	2.25	1.47
100000	-	0.80	4.04	2.46
1000000	-	0.91	4.96	3.90

Table 1: Average Query Time (μs) for Uniform Distribution

6 Conclusion

In this mini-project, we explored various hashing techniques with a focus on achieving worst-case constant search time. Starting from simple hashing approaches, we progressively implemented and evaluated more sophisticated methods, including universal hashing and perfect hashing. Our study included both 1-universal and 2-universal families to analyze collision behavior under randomized settings. We then implemented perfect hashing with quadratic space and extended it to a two-level perfect hashing scheme that achieves optimal space and constant-time lookups. Through empirical evaluation on both uniform and adversarial datasets, we demonstrated the trade-offs between simplicity, space efficiency, and worst-case performance. Overall, the project highlights how theoretical guarantees translate into practical performance, and reinforces the value of randomized algorithms in designing efficient data structures.