Hashing with Worst Case Constant Search Time

Tanuj Agarwal, Harsh Baid, Ayush
Group Number: 16

April 19, 2025

Outline

1. Introduction
2. Universal Hashing
3. Hashing with O(1) Search Time
4. Results and Analysis
5. Conclusion

Introduction

- Hashing is a widely used technique that allows storing and searching for elements in constant time on average. However, standard hash tables can suffer **from collisions**, which degrade performance—especially in the worst case.

Introduction

- Hashing is a widely used technique that allows storing and searching for elements in constant time on average. However, standard hash tables can suffer **from collisions**, which degrade performance—especially in the worst case.
- The only effective way to improve the situation is to choose the hash function **randomly**, independent of the keys to be stored.

Introduction

- Hashing is a widely used technique that allows storing and searching for elements in constant time on average. However, standard hash tables can suffer **from collisions**, which degrade performance—especially in the worst case.
- The only effective way to improve the situation is to choose the hash function **randomly**, independent of the keys to be stored.
- Our goal is to design hashing schemes that provide **worst-case constant-time (O(1))** lookups, regardless of the dataset.

Problem Definition

- Given a universe $U = \{1, 2, ..., m\}$ and subset $S \subseteq U$ of size s.
- Construct a structure to answer: *Does element i belong to S in $O(1)$?*
- $s \ll m$, e.g., $m = 10^{18}$, $s = 10^3$

Problem Definition

- Given a universe $U = \{1, 2, ..., m\}$ and subset $S \subseteq U$ of size s.
- Construct a structure to answer: *Does element i belong to S in $O(1)$?*
- $s \ll m$, e.g., $m = 10^{18}$, $s = 10^3$

Challenges:

- Cannot use a giant array of size m – memory is infeasible
- Binary search on a sorted array of S takes $O(\log s)$ time

Simple Hash Function: $i \mod n$

- Works well for uniformly random S.
- Low collision probability: $\mathbb{P}(h(i) = h(j)) \leq \frac{1}{n}$.
- Expected collisions: $O(1)$ for $n = O(s)$.

Simple Hash Function: $i \mod n$

- Works well for uniformly random S.
- Low collision probability: $\mathbb{P}(h(i) = h(j)) \leq \frac{1}{n}$.
- Expected collisions: $O(1)$ for $n = O(s)$.

Drawbacks: If a malicious adversary chooses the keys to be hashed by a fixed hash function like the one shown above, then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of $O(n)$.

Universal Hashing

- Fixed hash functions can be exploited by adversaries to cause many collisions, resulting in $O(n)$ worst-case search time.

Universal Hashing

- Fixed hash functions can be exploited by adversaries to cause many collisions, resulting in $O(n)$ worst-case search time.
- A collection **H** of hash functions is said to be **c-universal** if there exists a constant c such that for any $i, j \in U$,

$$\mathbb{P}(h(i) = h(j)) \leq \frac{c}{n}$$

where h is chosen uniformly at random from **H**.

Universal Hashing

- Fixed hash functions can be exploited by adversaries to cause many collisions, resulting in $O(n)$ worst-case search time.
- A collection **H** of hash functions is said to be **c-universal** if there exists a constant c such that for any $i, j \in U$,

$$\mathbb{P}(h(i) = h(j)) \leq \frac{c}{n}$$

  where h is chosen uniformly at random from **H**.
- Universal Hashing avoids $O(n)$ worst-case search time by randomly selecting the hash function from a family of hash functions, making it resistant to adversarial inputs.

2-Universal Hashing

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

$$\mathbb{P}(h(i) = h(j)) \leq \frac{2}{n}$$

where h is selected uniformly at random from **H**.

2-Universal Hashing

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

$$\mathbb{P}(h(i) = h(j)) \leq \frac{2}{n}$$

where h is selected uniformly at random from **H**.

- Let p be a prime number, $x \in \{1, 2, \ldots, p-1\}$.

$$h_x(i) = (ix \bmod p) \bmod n$$

The family of hash functions:

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

$\mathcal{H}$ is a 2-universal hash function family.

2-Universal Hashing

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

$$\mathbb{P}(h(i) = h(j)) \leq \frac{2}{n}$$

where h is selected uniformly at random from **H**.

- Let p be a prime number, $x \in \{1, 2, \ldots, p-1\}$.

  $$h_x(i) = (ix \bmod p) \bmod n$$

  The family of hash functions:

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

  $\mathcal{H}$ is a 2-universal hash function family.

- $\mathbb{P}(h(i) = h(j)) \leq \frac{2}{n}$

1-Universal Hashing

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

$$\mathbb{P}(h(i) = h(j)) \leq \frac{1}{n}$$

where h is selected uniformly at random from **H**.

1-Universal Hashing

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

$$\mathbb{P}(h(i) = h(j)) \leq \frac{1}{n}$$

where h is selected uniformly at random from **H**.

- Let p be a prime number, $x \in \{1, 2, \ldots, p-1\}$, $y \in \{0, 1, \ldots, p-1\}$.

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

The family of hash functions:

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

$\mathcal{H}$ is a 1-universal hash function family.

1-Universal Hashing

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

$$\mathbb{P}(h(i) = h(j)) \leq \frac{1}{n}$$

where h is selected uniformly at random from **H**.

- Let p be a prime number, $x \in \{1, 2, \ldots, p-1\}$, $y \in \{0, 1, \ldots, p-1\}$.

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

  The family of hash functions:

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

  $\mathcal{H}$ is a 1-universal hash function family.
- $\mathbb{P}(h(i) = h(j)) \leq \frac{1}{n}$

Algorithm

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 $T[h(i)] \neq \text{empty}$ **then**
      $t = 1$; **break**
    end if
  end for
until $t = 0$
return the perfect hash table T

Perfect Hashing in $O(s^2)$

Algorithm

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.

Two-Level Hashing ($O(s)$ Space)

Algorithm:

Fix $n = cs$
repeat
  Pick a random hash function $h \in_r \mathcal{H}$
  Set collision count $t \leftarrow 0$
  for all $i \in S$ **do**
    Compute $h(i)$
    $t \leftarrow t + \text{length}(T[h(i)])$
    Append i to list $T[h(i)]$
  end for
until $t \leq s$
Primary hash table is complete
for each $0 \leq i < n$ **do**
  if size of list $T[i] > 1$ **then**
    Build a perfect hash table for list $T[i]$
    Set $T[i]$ to point to this hash table
  end if
end for

Two-Level Hashing ($O(s)$ Space)

Algorithm:

Fix $n = cs$
repeat
    Pick a random hash function $h \in_r \mathcal{H}$
    Set collision count $t \leftarrow 0$
    for all $i \in S$ **do**
        Compute $h(i)$
        $t \leftarrow t + \text{length}(T[h(i)])$
        Append i to list $T[h(i)]$
    end for
until $t \leq s$
Primary hash table is complete
for each $0 \leq i < n$ **do**
    if size of list $T[i] > 1$ **then**
        Build a perfect hash table for list $T[i]$
        Set $T[i]$ to point to this hash table
    end if
end for

Analysis:

Time complexity:
Expected $O(s^2)$

Space complexity:
Expected $O(s)$

Search time:
Worst-case $O(1)$

Experimental Setup

Datasets Used:

- **Size Range:** Varied from small to very large datasets, specifically from 10^1 (10 elements) up to 10^8 (100 million elements).
- **Purpose:** To evaluate the performance and scalability of hashing algorithms under different data volumes.

Experimental Setup

Datasets Used:

- **Size Range:** Varied from small to very large datasets, specifically from 10^1 (10 elements) up to 10^8 (100 million elements).
- **Purpose:** To evaluate the performance and scalability of hashing algorithms under different data volumes.

Types of Input Distributions:

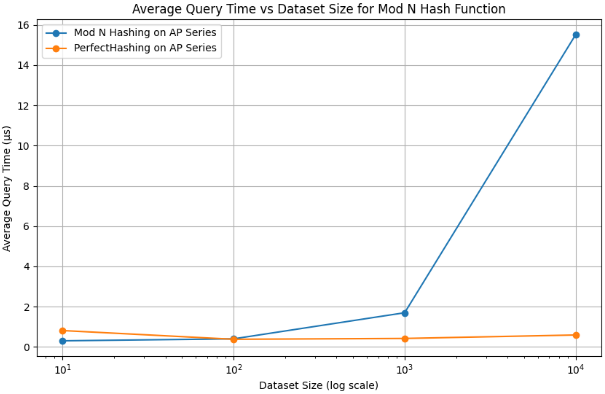
- **Uniform Distribution:** Keys are chosen randomly with equal probability — simulates unpredictable real-world data.
- **Arithmetic Progressions:** Structured sequences (e.g., 2, 4, 6, ...) to test performance on predictable and non-random data.

Results and Insights

- **Simple Hashing**: Fastest on uniform data, but fails catastrophically on structured data like AP distributions.

Results and Insights

- **Simple Hashing**: Fastest on uniform data, but fails catastrophically on structured data like AP distributions.

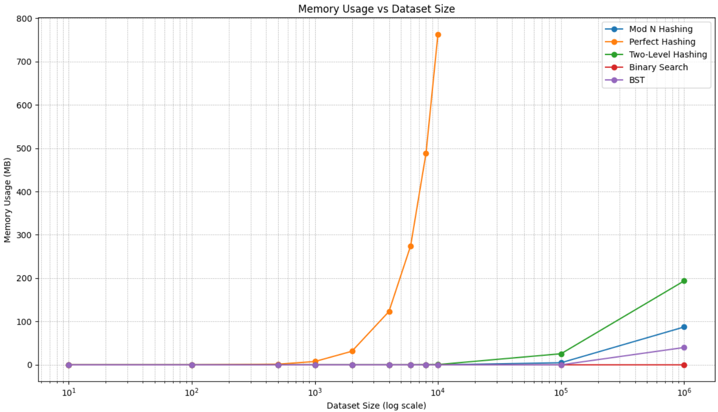


Results and Insights

- **Perfect Hashing**: Guaranteed constant-time lookups but memory overhead became unsustainable beyond moderate s.
- **Memory usage increases with dataset size:**

Results and Insights

- **Perfect Hashing**: Guaranteed constant-time lookups but memory overhead became unsustainable beyond moderate s.
- **Memory usage increases with dataset size:**

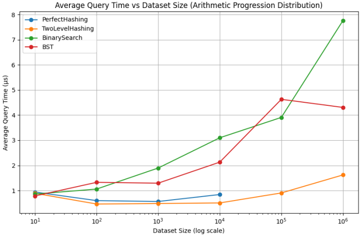


Results and Insights

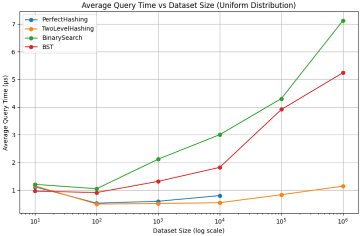
- **Two-Level Hashing**: Achieved the best of both worlds – fast lookups with high probability and space close to optimal.

Results and Insights

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



AP Distribution



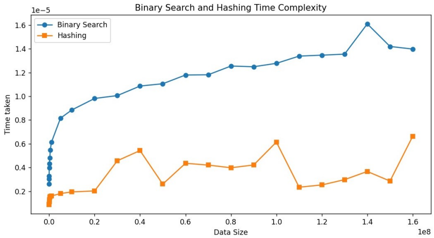
Uniform Distribution

Comparative Evaluation

- Compared performance of hashing with **binary search and Binary Search Trees (BSTs)**. Performance grew with $\log s$ in the latter – it was consistent but slower than hashing on larger datasets.

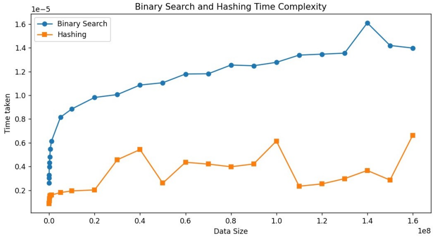
Comparative Evaluation

- Compared performance of hashing with **binary search and Binary Search Trees (BSTs)**. Performance grew with $\log s$ in the latter – it was consistent but slower than hashing on larger datasets.



Comparative Evaluation

- Compared performance of hashing with **binary search and Binary Search Trees (BSTs)**. Performance grew with $\log s$ in the latter – it was consistent but slower than hashing on larger datasets.



- Two-level hashing outperformed others in large-scale tests.

Empirical Results on Uniform Distribution

- **Empirical tests** measured average query time (in μs) across varying dataset sizes on **uniformly distributed data**.

Empirical Results on Uniform Distribution

- **Empirical tests** measured average query time (in μs) across varying dataset sizes on **uniformly distributed data**.
- **Perfect and two-level hashing** consistently outperformed **comparison-based methods**.

Empirical Results on Uniform Distribution

- **Empirical tests** measured average query time (in μs) across varying dataset sizes on **uniformly distributed data**.
- **Perfect and two-level hashing** consistently outperformed **comparison-based methods**.
- **Perfect hashing** is omitted beyond 10^4 due to **high memory overhead**.

Empirical Results on Uniform Distribution

- **Empirical tests** measured average query time (in μs) across varying dataset sizes on **uniformly distributed data**.
- **Perfect and two-level hashing** consistently outperformed **comparison-based methods**.
- **Perfect hashing** is omitted beyond 10^4 due to **high memory overhead**.
- **Two-level hashing** remained efficient and **scalable**, even for large datasets.

Empirical Results on Uniform Distribution

- **Empirical tests** measured average query time (in μs) across varying dataset sizes on **uniformly distributed data**.
- **Perfect and two-level hashing** consistently outperformed **comparison-based methods**.
- **Perfect hashing** is omitted beyond 10^4 due to **high memory overhead**.
- **Two-level hashing** remained efficient and **scalable**, even for large datasets.

Dataset Size	Perfect Hashing	Two-Level Hashing	Binary Search	BST
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

Conclusion

- Explored hashing techniques ensuring worst-case $O(1)$ search

Conclusion

- Explored hashing techniques ensuring worst-case $O(1)$ search
- Universal families reduce adversarial risk

Conclusion

- Explored hashing techniques ensuring worst-case $O(1)$ search
- Universal families reduce adversarial risk
- Two-level hashing balances time and space efficiently

Q&A

We are open to questions now!!