Performance Of Different Hashing Techniques:

Cuckoo Hashing vs Linear and Quadratic Probing

Team Members:
Syed Muhammad Wajeeh Haider
Shayaan Qazi
Muhammad Hassaan Tariq
Shayaan Asim

A project report presented for the course
Data Structures II

Department of Computer Science
HABIB UNIVERSITY

May 1, 2024

Contents

1 Introduction

1.1 Overview of Hashing

Hans Peter Luhn, a conscientious scientist at IBM, is credited with the invention of hashing. Working in the field of Computer Science and Information Science, Luhn devised hashing as a fundamental data structure for efficiently storing and retrieving data. Hashing involves mapping data to a specific index in a hash table using a hash function, enabling rapid access to information based on its key. This method finds widespread use in databases, caching systems, and various programming applications, facilitating optimized search and retrieval operations.

1.2 Importance of Efficient Hashing

Efficient hashing holds significant importance in computer science and software engineering due to several reasons:

- **Fast Retrieval:** Efficient hashing allows for quick retrieval of data by directly accessing the location where the data is stored in the hash table.
- **Reduced Search Time:** Hashing techniques minimize the time required to search for an element in a large dataset, leading to improved performance.
- **Space Optimization:** Properly designed hash functions and collision resolution techniques help optimize memory usage by minimizing collisions and ensuring even distribution of data in the hash table.
- **Data Security:** Hashing is essential for securing sensitive information through techniques like password hashing, ensuring that data remains protected even if the underlying storage is compromised.

Practical applications of efficient hashing include:

- **Database Indexing:** Hashing is commonly used in database management systems to create indexes for efficient data retrieval.
- **Caching Systems:** Hash tables are integral to caching systems, where they store frequently accessed data to improve application performance.
- **Cryptographic Operations:** Hash functions play a crucial role in cryptographic algorithms for generating message digests, digital signatures, and ensuring data integrity.
- **Network Routing:** Hashing is utilized in network routing algorithms to distribute traffic evenly across multiple paths, ensuring efficient utilization of network resources.

2 Cuckoo Hashing

Cuckoo hashing is a hash table-based data structure that provides a method for resolving collisions by using multiple hash functions and multiple hash tables. It was first introduced by Pagh and Rodler in 2001.

2.0.1 Mechanism

The key idea behind cuckoo hashing is to maintain two or more separate hash tables, each associated with a different hash function. When inserting a new key-value pair into the hash table, the key is hashed using each of the hash functions, and the corresponding hash table locations are checked.

If any of the hash table locations is unoccupied, the key-value pair is inserted into that location. However, if all the locations are occupied, cuckoo hashing employs a displacement strategy to resolve the collision. This strategy involves "kicking out" one of the existing key-value pairs from its location and inserting the new pair in its place.

The displaced key-value pair is then rehashed using the other hash function, and the process is repeated until a vacant position is found or a maximum number of displacement attempts is reached. If the latter occurs, the hash tables are resized and rehashed to accommodate the new key.

2.0.2 Operations

Insertion

Insertion in cuckoo hashing involves the following steps:

1. Hash the key using each of the hash functions.
2. Check the corresponding positions in each hash table.
3. If any position is unoccupied, insert the key-value pair into that position.
4. If all positions are occupied, use the displacement strategy to make room for the new key-value pair.
5. Repeat the process until the key is successfully inserted or a maximum displacement threshold is reached.

Hash Functions:

```
h1(key) = key%11
h2(key) = (key/11)%11
```

	20	50	53	75	100	67	105	3	36	39
h_1(key)	9	6	9	9	1	1	6	3	3	6
h_2(key)	1	4	4	6	9	6	9	0	3	3

Figure 1: Hash Functions and Hasing Tables

Let's start by inserting **20** at its possible position in the first table determined by h1(20):

table[1]	-	-	-	-	-	-	-	-	-	20	-
table[2]	-	-	-	-	-	-	-	-	-	-	-

Figure 2: Inserting 20 in Table 1

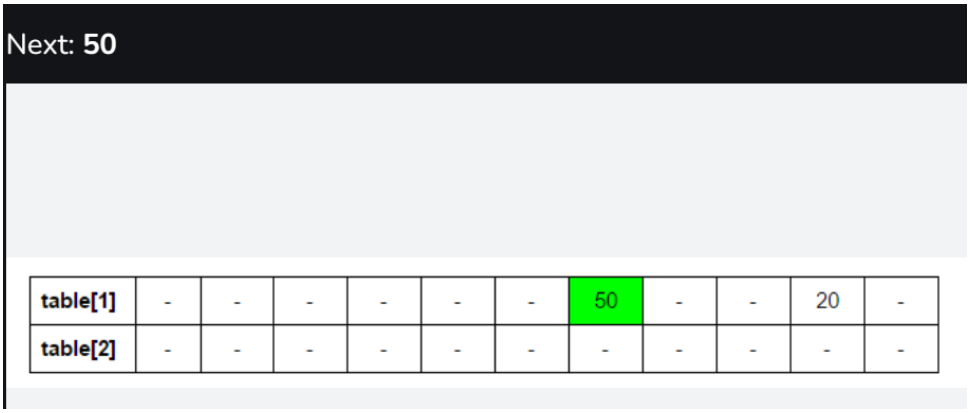

Figure 3: Inserting 50 in Table 1

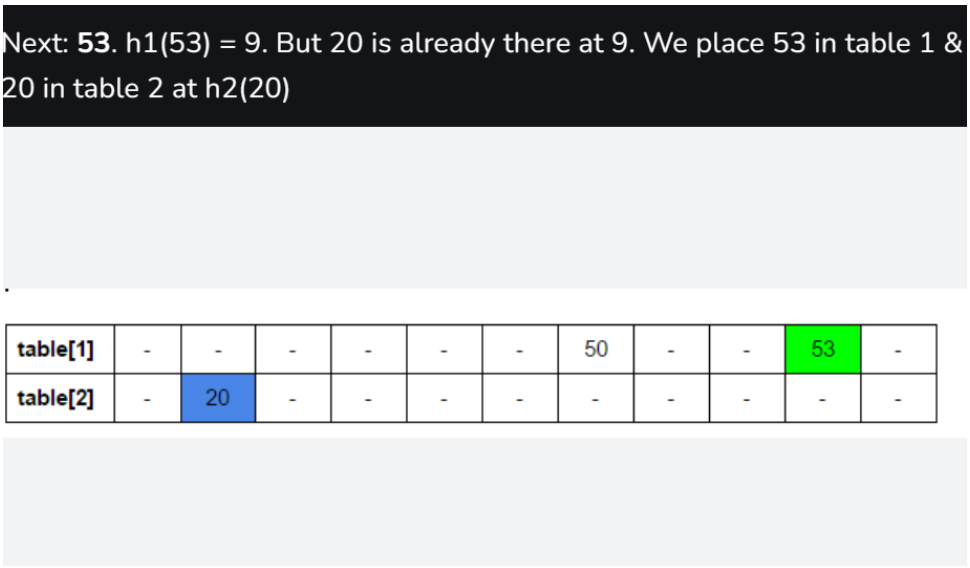

Figure 4: Insert 53

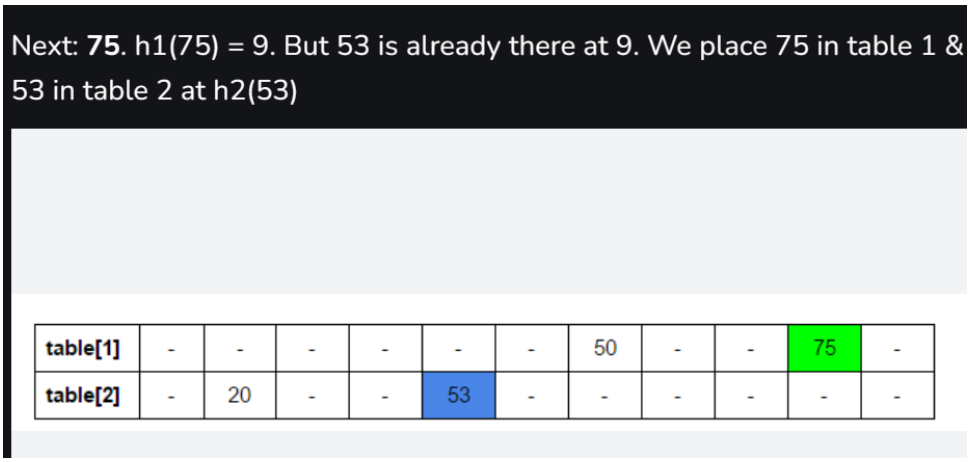

Figure 5: Insert 75

Next: **100**. h1(100) = 1.

table[1]	-	100	-	-	-	-	50	-	-	75	-
table[2]	-	20	-	-	53	-	-	-	-	-	-

Figure 6: Insert 100

Next: **67**. h1(67) = 1. But 100 is already there at 1. We place 67 in table 1 & 100 in table 2

table[1]	-	67	-	-	-	-	50	-	-	75	-
table[2]	-	20	-	-	53	-	-	-	-	100	-

Figure 7: Insert 67

Next: **105**. h1(105) = 6. But 50 is already there at 6. We place 105 in table 1 & 50 in table 2 at h2(50) = 4. Now 53 has been displaced. h1(53) = 9. 75 displaced: h2(75) = 6.

table[1]	-	67	-	-	-	-	105	-	-	53	-
table[2]	-	20	-	-	50	-	75	-	-	100	-

Figure 8: Insert 105

Next: **3**. h1(3) = 3.

table[1]	-	67	-	3	-	-	105	-	-	53	-
table[2]	-	20	-	-	50	-	75	-	-	100	-

Figure 9: Insert 3

Next: **36**. h1(36) = 3. h2(3) = 0.

table[1]	-	67	-	36	-	-	105	-	-	53	-
table[2]	3	20	-	-	50	-	75	-	-	100	-

Figure 10: Insert 36

Next: **39**. h1(39) = 6. h2(105) = 9. h1(100) = 1. h2(67) = 6. h1(75) = 9. h2(53) = 4. h1(50) = 6. h2(39) = 3.
Here, the new key 39 is displaced later in the recursive calls to place 105, which it displaced.

table[1]	-	100	-	36	-	-	50	-	-	75	-
table[2]	3	20	-	39	53	-	67	-	-	105	-

Figure 11: Insert 39

Lookup
Lookup operation in cuckoo hashing is similar to insertion:

1. Hash the key using each of the hash functions.
2. Check the corresponding positions in each hash table.
3. If the key is found in any position, return the corresponding value.
4. If the key is not found, return a "not found" indication.

Deletion
Deletion operation in cuckoo hashing involves the following steps:

1. Hash the key using each of the hash functions.
2. Check the corresponding positions in each hash table.
3. If the key is found in any position, remove the key-value pair from that position.
4. If the key is not found, do nothing.

2.1 Advantages

Cuckoo hashing offers several advantages over traditional collision resolution techniques:

- **Constant-Time Operations:** In ideal conditions, cuckoo hashing provides constant-time insertion, deletion, and lookup operations, making it highly efficient for large datasets.
- **High Load Factor Tolerance:** Cuckoo hashing can handle high load factors without significant degradation in performance, as long as the hash functions distribute the keys evenly across the hash tables.
- **Deterministic Performance:** Unlike some other collision resolution methods, cuckoo hashing provides deterministic performance guarantees, ensuring consistent behavior regardless of the dataset.

2.2 Limitations

Despite its advantages, cuckoo hashing has some limitations:

- **Space Overhead:** Cuckoo hashing requires multiple hash tables, leading to increased memory overhead compared to single-table approaches.
- **Complexity:** Implementing cuckoo hashing can be more complex than traditional collision resolution techniques, particularly when designing and managing multiple hash functions and tables.
- **Performance Sensitivity:** Cuckoo hashing performance is sensitive to the choice of hash functions and the initial hash table size, requiring careful tuning for optimal results.

2.3 Applications

Cuckoo hashing finds applications in various domains where fast and efficient lookup operations are critical, including:

- **In-Memory Databases:** Cuckoo hashing is used in in-memory databases to store and retrieve data with minimal latency.
- **Networking:** Cuckoo hashing is employed in networking devices for route lookup and packet forwarding, ensuring high-speed data transmission.
- **Caching Systems:** Cuckoo hashing is utilized in caching systems to quickly access frequently requested data and improve application performance.
- **Parallel and Distributed Computing:** Cuckoo hashing is suitable for parallel and distributed computing environments where fast and scalable data access is essential.

Overall, cuckoo hashing provides a robust and efficient solution for handling collisions in hash tables, offering deterministic performance and constant-time operations under favorable conditions.

3 Our Implementations

3.1 Hashing Techniques Used

Hashing techniques are designed to convert a range of key values into a range of indexes of an array. We have taken into account 3 such techniques;

- Cuckoo Hashing
- Linear Probing
- Quadratic Probing

3.1.1 Cuckoo Hashing

Cuckoo Hashing as discussed in Section 2 is a method to resolve collisions through the means of multiple hash tables and functions. The most important feature of Cuckoo Hashing its use of maintaining 2 hash tables instead of 1.

3.1.2 Linear Probing

Linear probing is a collision resolving technique in open addressing hash tables that places the new key into the closest following empty cell when a collision occurs.

3.1.3 Quadratic Probing

Quadratic probing is a collision resolving technique in hash tables that uses a form of quadratic polynomial to calculate the interval between probes.

3.2 Implemented Data Structures and Operations

In this section, we will delve into the specifics of how each hashing technique is implemented and the operations it supports.

3.2.1 Cuckoo Hashing Implementation

Cuckoo hashing involves two hash functions and two tables. The key is first hashed using the first hash function. If the slot is empty, the key is inserted; otherwise, the existing key is displaced and rehashed with the second hash function. Our Cuckoo Hashing is comprised of 3 functions, Insert, Look-up(Search) and Delete.

3.2.2 INSERTION

This function inserts a movie entry into the Cuckoo hash table. Here's how it works:

- It calculates two hash values (hash1 and hash2) for the movie name using two different hash functions.
- It checks if the slot at hash1 is empty. If it is, it inserts the entry into that slot in table1.
- If the slot at hash1 is not empty, it checks if the slot at hash2 is empty. If it is, it inserts the entry into that slot in table2.
- If both slots are occupied, it performs cuckoo hashing by repeatedly swapping the existing entry with the new entry until an empty slot is found. This ensures that both hash tables remain balanced.
- If the loop completes without finding an empty slot, it triggers a resizing operation to increase the size of the hash table.
- The function then recursively calls itself to insert the entry into the appropriate slot after resizing.

```
void CuckooHashTable::insert(const MovieEntry2& entry) {
    // Calculate hash values for the movie name using both hash functions
    size_t hash1 = hashFunction1(entry.Name) % size;
    size_t hash2 = hashFunction2(entry.Name) % size;

    // Check if the slot at hash1 is empty
    if (table1[hash1].Name.empty()) {
        table1[hash1] = entry;
        return;
    }
    // Check if the slot at hash2 is empty
    if (table2[hash2].Name.empty()) {
        table2[hash2] = entry;
        return;
    }

    // If both slots are occupied, perform cuckoo hashing
    MovieEntry2 temp = entry;
    for (int i = 0; i < log(size); i++) {
        // Swap the entry with the slot at hash1 on even iterations
        if (i % 2 == 0) {
            std::swap(temp, table1[hash1]);
            hash1 = hashFunction1(temp.Name) % size;
            if (table1[hash1].Name.empty()) {
                table1[hash1] = temp;
                return;
            }
        }
```

Figure 12: Insert Function

```
        // Swap the entry with the slot at hash2
        else {
            std::swap(temp, table2[hash2]);
            hash2 = hashFunction2(temp.Name) % size;
            if (table2[hash2].Name.empty()) {
                table2[hash2] = temp;
                return;
            }
        }
    }

    // If the loop completes, the hash table is full
    if (totalentries > 0.75*size){
        std::cerr << "Resizing triggered." << entry.Name << "  Size:" << size << std::endl;
        resize();
    }
    insert(temp);
}
```

Figure 13: Insert Function

3.2.3 Search

This function searches for a movie entry in the Cuckoo hash table. Here's how it works:

- It calculates two hash values (`hash1` and `hash2`) for the movie name using the same hash functions used during insertion.
- It checks if the movie is stored at `hash1`. If it is, it returns a pointer to the movie entry stored in `table1`.
- If the movie is not found at `hash1`, it checks if it is stored at `hash2`. If it is, it returns a pointer to the movie entry stored in `table2`.
- If the movie is not found in either table, it returns a `nullptr`, indicating that the movie is not present in the hash table.

```
MovieEntry2* CuckooHashTable::search(const string& key) {
    // Calculate hash values for the movie name using both hash functions
    size_t hash1 = hashFunction1(key) % size;
    size_t hash2 = hashFunction2(key) % size;

    // Check if the movie is stored at hash1
    if (table1[hash1].Name == key) {
        return &table1[hash1];
    }
    // Check if the movie is stored at hash2
    if (table2[hash2].Name == key) {
        return &table2[hash2];
    }
    // Movie not found
    return nullptr;
}
```

Figure 14: Search Function

3.2.4 Delete

This function deletes a movie entry from the Cuckoo hash table. Here's how it works:

- It calculates two hash values (`hash1` and `hash2`) for the movie name using the same hash functions used during insertion.
- It checks if the movie is stored at `hash1`. If it is, it clears the slot at `hash1` in `table1` by assigning it an empty `MovieEntry2` object.
- If the movie is not found at `hash1`, it checks if it is stored at `hash2`. If it is, it clears the slot at `hash2` in `table2` by assigning it an empty `MovieEntry2` object.
- If the movie is not found in either table, it prints an error message indicating that the movie was not found in the hash table.

```
void CuckooHashTable::deleteEntry(const string& key) {
    // Calculate hash values for the movie name using both hash functions
    size_t hash1 = hashFunction1(key) % size;
    size_t hash2 = hashFunction2(key) % size;

    // Check if the movie is stored at hash1
    if (table1[hash1].Name == key) {
        table1[hash1] = MovieEntry2();
        totalentries--;
        return;
    }
    // Check if the movie is stored at hash2
    if (table2[hash2].Name == key) {
        table2[hash2] = MovieEntry2();
        totalentries--;
        return;
    }
    // Movie not found
    std::cerr << "Movie not found." << std::endl;
}
```

Figure 15: Delete Function

3.3 Linear Probing Implementation

In linear probing, when a collision occurs, the algorithm checks the next cell in the array until an empty cell is found.

3.3.1 Insert

The `insert` function inserts a new movie entry into the linear probing hash table. Here's how it works:

- It calculates the hash value for the movie name using the hash function.

- It performs linear probing to find an empty slot in the hash table.
- Once an empty slot is found, it inserts the movie entry into that slot.

```
void LinearProbeHashTable::insert(const MovieEntry2& entry) {
    // Calculating hash value for the movie name
    size_t index = hashFunction(entry.Name);

    // Linear probing to find an empty slot
    while (!table[index].Name.empty() and table[index].Name != "tombstone") {
        index = (index + 1) % capacity;
    }

    // Inserting the entry at the found index
    table[index] = entry;
    size++;
}
```

Figure 16: Linear Probing Insert Function

3.3.2 Search

The `search` function searches for a movie entry in the linear probing hash table. Here's how it works:

- It calculates the hash value for the given key using the hash function.
- It iterates through the hash table using linear probing until it finds the entry with the matching key or encounters an empty slot.
- If it finds the entry with the matching key, it returns a pointer to that entry.
- If it encounters an empty slot before finding the entry, it returns `nullptr`, indicating that the entry is not present in the hash table.

```
MovieEntry2* LinearProbeHashTable::search(const string& key) {
    // Calculating hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    while (!table[index].Name.empty()) {
        if (table[index].Name == key) {
            return &table[index];
        }

        // Linear probing
        index = (index + 1) % capacity;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If key is not found
    return nullptr;
}
```

Figure 17: Linear Probing Search Function

3.3.3 Delete

The `deleteEntry` function deletes a movie entry from the linear probing hash table. Here's how it works:

- It calculates the hash value for the given key using the hash function.
- It iterates through the hash table using linear probing until it finds the entry with the matching key or encounters an empty slot.
- If it finds the entry with the matching key, it marks that slot as deleted by assigning the value "tombstone" to the `Name` field of the entry.
- If it encounters an empty slot before finding the entry, it prints an error message indicating that the entry with the given key was not found in the hash table.

```
void LinearProbeHashTable::deleteEntry(const std::string& key) {
    // Calculate hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    // Linear probing to find the entry with the given key
    while (!table[index].Name.empty()) {
        if (table[index].Name == key) {
            // Found the entry, mark it as deleted (Tombstone)
            table[index].Name = "tombstone";

            size--;
            return;
        }

        // Linear Probing
        index = (index + 1) % capacity;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If the loop ends without finding the entry, it's not present in the table
    std::cerr << "Entry with key '" << key << "' not found." << std::endl;
}
```

Figure 18: Linear Probing Delete Function

3.4 Quadratic Probing Implementation

Quadratic probing uses a hash function with a quadratic formula to calculate the probe sequence. Unlike linear probing, the interval between probes increases quadratically.

3.4.1 Insert

The `insert` function inserts a new movie entry into the quadratic probing hash table. Here's how it works:

- It calculates the hash value for the movie name using the hash function.
- It performs quadratic probing to find an empty slot in the hash table.
- It uses a quadratic sequence to determine the next slot to check, increasing the offset quadratically with each iteration.
- Once an empty slot is found, it inserts the movie entry into that slot.

```
void QuadraticProbeHashTable::insert(const MovieEntry2& entry) {
    // Calculating hash value for the movie name
    size_t index = hashFunction(entry.Name);

    // Quadratic probing to find an empty slot
    int i = 1;
    while (!table[index].Name.empty() and table[index].Name != "tombstone") {
        index = (index + i * i) % capacity;
        i++;
    }

    // Inserting the entry at the found index
    table[index] = entry;
    size++;
}
```

Figure 19: Quadratic Probing Insert Function

3.4.2 Search

The `search` function searches for a movie entry in the quadratic probing hash table. Here's how it works:

- It calculates the hash value for the given key using the hash function.
- It iterates through the hash table using quadratic probing until it finds the entry with the matching key or encounters an empty slot.
- It uses a quadratic sequence to determine the next slot to check, increasing the offset quadratically with each iteration.
- If it finds the entry with the matching key, it returns a pointer to that entry.
- If it encounters an empty slot before finding the entry, it returns `nullptr`, indicating that the entry is not present in the hash table.

```
MovieEntry2* QuadraticProbeHashTable::search(const string& key) {
    // Calculating hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    int i = 1;
    while (!table[index].Name.empty()) {

        if (table[index].Name == key) {
            return &table[index];
        }

        // Quadratic probing
        index = (index + i * i) % capacity;
        i++;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If key is not found
    return nullptr;
}
```

Figure 20: Quadratic Probing Search Function

3.4.3 Delete

The `deleteEntry` function deletes a movie entry from the quadratic probing hash table. Here's how it works:

- It calculates the hash value for the given key using the hash function.
- It iterates through the hash table using quadratic probing until it finds the entry with the matching key or encounters an empty slot.
- It uses a quadratic sequence to determine the next slot to check, increasing the offset quadratically with each iteration.
- If it finds the entry with the matching key, it marks that slot as deleted by assigning the value "tombstone" to the `Name` field of the entry.
- If it encounters an empty slot before finding the entry, it prints an error message indicating that the entry with the given key was not found in the hash table.

```
void QuadraticProbeHashTable::deleteEntry(const std::string& key) {
    // Calculate hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    // Quadratic probing to find the entry with the given key
    int i = 1;
    while (!table[index].Name.empty()) {
        if (table[index].Name == key) {
            // Found the entry, mark it as deleted (Tombstone)
            table[index].Name = "tombstone";

            size--;
            return;
        }

        index = (index + i * i) % capacity;
        i++;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If the loop ends without finding the entry, it's not present in the table
    std::cerr << "Entry with key '" << key << "' not found." << std::endl;
}
```

Figure 21: Quadratic Probing Delete Function

3.5 Motivation, Strengths, and Weaknesses

In this section, we will look into the motivation behind the development of each hashing technique, exploring their respective strengths, advantages, and limitations. Understanding these aspects is crucial for selecting the most appropriate hashing technique for various applications and scenarios.

3.5.1 Cuckoo Hashing:

Motivation: Cuckoo hashing emerged as a response to the growing demand for efficient collision resolution in hash tables. Traditional methods often suffered from performance degradation under high load factors or in the presence of clustering. Cuckoo hashing sought to address these issues by leveraging multiple hash functions and tables, offering a worst-case constant lookup time.

3.5.2 Linear Probing:

Motivation: Linear probing was developed as a simple and straightforward method for resolving collisions in hash tables. Its linear search approach allows for easy implementation and low overhead, making it suitable for scenarios where simplicity and efficiency are paramount.
Strengths:

- **Simplicity:** Linear probing is easy to implement and understand, requiring minimal additional data structures or computations.
- **Low Overhead:** Linear probing typically incurs low memory overhead, as it only requires additional space for the hash table itself.
- **Cache-Friendly:** Linear probing exhibits good cache locality, which can lead to better performance in practice, especially for small to moderate-sized datasets.

Weaknesses:

- **Primary Clustering:** Linear probing is prone to primary clustering, where consecutive collisions lead to longer probe sequences and potential performance degradation.
- **Secondary Clustering:** Even though linear probing avoids primary clustering, it is susceptible to secondary clustering, where nearby clusters fill up, leading to longer search times.
- **Performance Variability:** The performance of linear probing can vary significantly depending on the dataset and the hash function used, making it less predictable in some cases.

3.5.3 Quadratic Probing:

Motivation: Quadratic probing was developed to mitigate the primary clustering issue observed in linear probing. By using a quadratic polynomial to calculate probe sequences, it aims to distribute collided keys more evenly throughout the hash table, reducing the

likelihood of clustering.
Strengths:

- **Reduced Clustering:** Quadratic probing reduces primary clustering compared to linear probing, leading to more even distribution of collided keys and potentially better performance.
- **Ease of Implementation:** Similar to linear probing, quadratic probing is relatively simple to implement and understand, requiring only basic arithmetic operations.
- **Performance Stability:** Quadratic probing tends to exhibit more stable performance across different datasets and hash functions compared to linear probing, making it a more predictable option in some scenarios.

Weaknesses:

- **Secondary Clustering:** While quadratic probing reduces primary clustering, it may still suffer from secondary clustering under certain conditions, leading to performance degradation.
- **Increased Probe Sequence Length:** Quadratic probing can lead to longer probe sequences compared to linear probing, especially as the load factor increases, potentially impacting performance.
- **Quadratic Polynomial Overhead:** Calculating the probe sequence using a quadratic polynomial introduces additional computational overhead, although it is typically negligible for small to moderate-sized hash tables.

4 Time Complexity Analysis

4.1 Cuckoo Hasing

Let's analyze the time complexities of the `insert`, `delete`, and `search` operations performed in our code. You can refer to the snippets of the codes below under their respective sections

4.1.1 Insertion Time Complexity

The insertion operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the slots at the calculated hash values are empty.
3. If the slots are occupied, performing cuckoo hashing.

In the `insert` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.

- Checking if slots are empty and assigning entries are constant time operations.
- The cuckoo hashing loop iterates at most log(size) times.

Therefore, the time complexity of insertion is $O(\log \text{size})$.

```
MovieEntry2* QuadraticProbeHashTable::search(const string& key) {
    // Calculating hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    int i = 1;
    while (!table[index].Name.empty()) {

        if (table[index].Name == key) {
            return &table[index];
        }

        // Quadratic probing
        index = (index + i * i) % capacity;
        i++;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If key is not found
    return nullptr;
}
```

Figure 22: Quadratic Probing Search Function

4.1.2 Deletion Time Complexity

The deletion operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the entry exists at the calculated hash values and deleting it if found.

In the `deleteEntry` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.
- Checking if the entry exists and deleting it are constant time operations.

Therefore, the time complexity of deletion is $O(1)$.

```
void CuckooHashTable::deleteEntry(const string& key) {
    // Calculate hash values for the movie name using both hash functions
    size_t hash1 = hashFunction1(key) % size;
    size_t hash2 = hashFunction2(key) % size;

    // Check if the movie is stored at hash1
    if (table1[hash1].Name == key) {
        table1[hash1] = MovieEntry2();
        totalentries--;
        return;
    }
    // Check if the movie is stored at hash2
    if (table2[hash2].Name == key) {
        table2[hash2] = MovieEntry2();
        totalentries--;
        return;
    }
    // Movie not found
    std::cerr << "Movie not found." << std::endl;
}
```

Figure 23: Cuckoo hashing delete function

4.1.3 Search Time Complexity

The search operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the entry exists at the calculated hash values.

In the `search` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.
- Checking if the entry exists is a constant time operation.

Therefore, the time complexity of search is $O(1)$.

```
MovieEntry2* CuckooHashTable::search(const string& key) {
    // Calculate hash values for the movie name using both hash functions
    size_t hash1 = hashFunction1(key) % size;
    size_t hash2 = hashFunction2(key) % size;

    // Check if the movie is stored at hash1
    if (table1[hash1].Name == key) {
        return &table1[hash1];
    }
    // Check if the movie is stored at hash2
    if (table2[hash2].Name == key) {
        return &table2[hash2];
    }
    // Movie not found
    return nullptr;
}
```

Figure 24: Cuckoo hashing search function

Based on the analysis of the provided code, the time complexities are as follows:

- Insertion: $O(\log \text{size})$
- Deletion: $O(1)$
- Search: $O(1)$

References: The analysis is based on the our C++ code for cuckoo hashing. you can refer to the images for the insert, lookup and deletion code provided above

4.2 Linear Probing

4.2.1 Insertion Time Complexity

The insertion operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the slots at the calculated hash values are empty.
3. If the slots are occupied, performing linear probing to find an empty slot.

In the `insert` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.
- Checking if slots are empty and assigning entries are constant time operations.
- Linear probing iterates through slots until an empty slot is found.

Therefore, the time complexity of insertion is $O(1)$.

```
void LinearProbeHashTable::insert(const MovieEntry2& entry) {
    // Calculating hash value for the movie name
    size_t index = hashFunction(entry.Name);

    // Linear probing to find an empty slot
    while (!table[index].Name.empty() and table[index].Name != "tombstone") {
        index = (index + 1) % capacity;
    }

    // Inserting the entry at the found index
    table[index] = entry;
    size++;
}
```

Figure 25: Linear Probing Insert Function

4.2.2 Deletion Time Complexity

The deletion operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the entry exists at the calculated hash values and deleting it if found.

In the `deleteEntry` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.
- Checking if the entry exists and deleting it are constant time operations.

Therefore, the time complexity of deletion is $O(1)$.

```cpp
void LinearProbeHashTable::deleteEntry(const std::string& key) {
    // Calculate hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    // Linear probing to find the entry with the given key
    while (!table[index].Name.empty()) {
        if (table[index].Name == key) {
            // Found the entry, mark it as deleted (Tombstone)
            table[index].Name = "tombstone";

            size--;
            return;
        }

        // Linear Probing
        index = (index + 1) % capacity;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If the loop ends without finding the entry, it's not present in the table
    std::cerr << "Entry with key '" << key << "' not found." << std::endl;
}
```

Figure 26: Linear Probing Delete Function

4.2.3 Search Time Complexity

The search operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the entry exists at the calculated hash values.

In the `search` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.
- Checking if the entry exists is a constant time operation.

Therefore, the time complexity of search is $O(1)$.

```
MovieEntry2* LinearProbeHashTable::search(const string& key) {
    // Calculating hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    while (!table[index].Name.empty()) {
        if (table[index].Name == key) {
            return &table[index];
        }

        // Linear probing
        index = (index + 1) % capacity;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If key is not found
    return nullptr;
}
```

Figure 27: Linear Probing Search Function

4.3 Quadratic Probing

4.3.1 Insertion Time Complexity

The insertion operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the slots at the calculated hash values are empty.
3. If the slots are occupied, performing quadratic probing to find an empty slot.

In the `insert` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.
- Checking if slots are empty and assigning entries are constant time operations.
- Quadratic probing iterates through slots until an empty slot is found.

Therefore, the time complexity of insertion is $O(1)$.

```
void QuadraticProbeHashTable::insert(const MovieEntry2& entry) {
    // Calculating hash value for the movie name
    size_t index = hashFunction(entry.Name);

    // Quadratic probing to find an empty slot
    int i = 1;
    while (!table[index].Name.empty() and table[index].Name != "tombstone") {
        index = (index + i * i) % capacity;
        i++;
    }

    // Inserting the entry at the found index
    table[index] = entry;
    size++;
}
```

Figure 28: Quadratic Probing Insert Function

4.3.2 Deletion Time Complexity

The deletion operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the entry exists at the calculated hash values and deleting it if found.

In the `deleteEntry` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.
- Checking if the entry exists and deleting it are constant time operations.

Therefore, the time complexity of deletion is $O(1)$.

```
void QuadraticProbeHashTable::deleteEntry(const std::string& key) {
    // Calculate hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    // Quadratic probing to find the entry with the given key
    int i = 1;
    while (!table[index].Name.empty()) {
        if (table[index].Name == key) {
            // Found the entry, mark it as deleted (Tombstone)
            table[index].Name = "tombstone";

            size--;
            return;
        }

        index = (index + i * i) % capacity;
        i++;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If the loop ends without finding the entry, it's not present in the table
    std::cerr << "Entry with key '" << key << "' not found." << std::endl;
}
```

Figure 29: Quadratic Probing Delete Function

4.3.3 Search Time Complexity

The search operation involves the following steps:

1. Calculating hash values for the entry name using both hash functions.
2. Checking if the entry exists at the calculated hash values.

In the `search` function:

- Calculating hash values using `hashFunction1` and `hashFunction2` takes constant time.
- Checking if the entry exists is a constant time operation.

Therefore, the time complexity of search is $O(1)$.

```
MovieEntry2* QuadraticProbeHashTable::search(const string& key) {
    // Calculating hash value for the key
    size_t index = hashFunction(key);

    // Original index to check if we have looped back
    int originalIndex = index;

    int i = 1;
    while (!table[index].Name.empty()) {

        if (table[index].Name == key) {
            return &table[index];
        }

        // Quadratic probing
        index = (index + i * i) % capacity;
        i++;

        // If we've looped back to the original index, the key is not present
        if (index == originalIndex) {
            break;
        }
    }

    // If key is not found
    return nullptr;
}
```

Figure 30: Quadratic Probing Search Function

Based on the analysis of the provided code, the time complexities are as follows:

- Linear Probing Insertion, Deletion, Search: $O(1)$
- Quadratic Probing Insertion, Deletion, Search: $O(1)$

References: The analysis is based on our C++ code for linear probing and quadratic probing. You can refer to the images for the insert, lookup, and deletion code provided above.

5 Application and Demo

5.1 Sample Application Description

We describe a sample application that utilizes hashing techniques:

The sample application is a movie database management system that employs various hashing techniques for efficient data storage and retrieval. The system allows users to store, search, and delete movie entries using different hash tables, including cuckoo hashing, linear probing, and quadratic probing. The application loads movie data from a CSV file, parses the data, and inserts it into the hash tables. Users can then search for movies by name, year, rating, or other attributes, and delete entries if necessary. This demonstration showcases the effectiveness of hashing techniques in managing large datasets efficiently.

5.2 Demonstration of Features

We demonstrate features such as inserting elements, searching elements, and deleting elements:

The demonstration of the sample application features includes the following steps:

1. Inserting elements: The application inserts movie entries from a CSV file into the hash tables using different hashing techniques, including cuckoo hashing, linear probing, and quadratic probing.
2. Searching elements: Users can search for movie entries by name, year, rating, or other attributes using the hash tables. The application demonstrates efficient searching capabilities provided by the hashing techniques.
3. Deleting elements: Users have the option to delete movie entries from the hash tables. The application showcases the ability to remove entries while maintaining data integrity and efficiency.

References

[1] Netflix Dataset. (2024, April 19). Kaggle. https://www.kaggle.com/datasets/paramvir705/netflix-dataset

[2] Cuckoo Hashing. GeeksforGeeks. https://www.geeksforgeeks.org/cuckoo-hashing/