HASHING: Hashing Functions, Collision Resolution, Deletion, and Perfect Hashing Function

In Partial fulfillment of the requirements in

CCE105/L: Data Structure

Submitted by:

JERRYSON T. ARBOL

CRIS JUNE T. CABARDO

STEPHEN JOHN T. CAMPILAN

HAZEL MAY T. RACHO

Submitted to:

CYVIL DAVE T. DASARGO, MIT

Table of Contents

Page

Hashing is a technique used to convert a given key into a fixed-size string of characters, which is typically a hash code. This code is used to index data in hash tables, allowing for fast data retrieval. The core idea is to use a hash function to map keys to indices in an array.

Hash Function

Hash Function: The hash() method computes the index for a given key using the built-in hashCode() method and ensures it stays within the bounds of the array. There are different methods of hash function such as;

1. Division Method (K mod m)

The division method for hashing involves dividing the key value by a prime number (usually close to the size of the hash table) and using the remainder as the hash value.

Example:

- Hash table size: 11 (a prime number)

- Key: 25

- Hash value = 25 % 11 = 3

- Thus, the key 25 is stored in index 3 of the hash table.

```java
class DivisionHash {
    public static int hash(int key, int tableSize) {
        return key % tableSize;
    }

    public static void main(String[] args) {
        int key = 25;
        int tableSize = 11;
        System.out.println("Division Hash Index: " + hash(key, tableSize));
    }
}
```

Figure 1: Division Method Code

2. Mid-Square Method (K^2 mod m)

The mid-square method involves squaring the key and then taking the middle part of the resulting number to generate the hash value. The selected middle digits are then converted to an index within the hash table.

Example:

- Key: 1234

- Square of the key: (1234^2 = 1522756)

- Middle part: Let's take the 2 middle digits (from the total of 7 digits, we take the 3rd and 4th), which gives us 22.

- Hash table size: 10

- Hash value = 22 % 10 = 2

- Thus, the key 1234 is stored in index 2 of the hash table.

```java
class MidSquareHash {
    public static int hash(int key, int tableSize) {
        int squared = key * key;
        String squaredStr = String.valueOf(squared);

        // Get the middle two digits
        int midIndex = squaredStr.length() / 2;
        String midDigits = squaredStr.substring(midIndex - 1, midIndex + 1); // Middle two

        return Integer.parseInt(midDigits) % tableSize;
    }

    public static void main(String[] args) {
        int key = 1234;
        int tableSize = 10;
        System.out.println("Mid-Square Hash Index: " + hash(key, tableSize));
    }
}
```

Figure 2: Mid Square Method Code

3. Folding Method ($K^1 + K^2 + K^3 \ldots K_n$)

The folding method involves dividing the key into equal parts, summing those parts, and then using the sum to compute the hash value. If the key cannot be divided evenly, the last part may be shorter.

Example:

- Key: 123456

- Split into parts: 123 and 456

- Sum of parts: 123 + 456 = 579

- Hash table size: 10

- Hash value = 579 % 10 = 9

- Thus, the key 123456 is stored in index 9 of the hash table.

```java
class FoldingHash {
    public static int hash(int key, int tableSize) {
        String keyStr = String.valueOf(key);
        int sum = 0;

        // Split key into parts of 2 digits each
        for (int i = 0; i < keyStr.length(); i += 2) {
            String part = keyStr.substring(i, Math.min(i + 2, keyStr.length()));
            sum += Integer.parseInt(part);
        }

        return sum % tableSize;
    }

    public static void main(String[] args) {
        int key = 123456;
        int tableSize = 10;
        System.out.println("Folding Hash Index: " + hash(key, tableSize));
    }
}
```

Figure 3: Folding Method Code

4. Radix Transformation (Key = (Remainder0 * Base^0) + (Remainder1 * Base^1) + ... + (RemainderN * Base^N))

The radix transformation method involves representing the key in a different base (radix) and using the digits in that base to compute a hash value.

Example:

- Key: 10 (in decimal)

- Convert to binary (base 2): 10 (decimal) = 1010 (binary)

- Sum the binary digits: 1 + 0 + 1 + 0 = 2

- Hash table size: 5

- Hash value = 2 % 5 = 2

- Thus, the key 10 is stored in index 2 of the hash table.

```java
class RadixTransformationHash {
    public static int hash(int key, int tableSize) {
        String binaryStr = Integer.toBinaryString(key);
        int sum = 0;

        // Sum the binary digits
        for (char bit : binaryStr.toCharArray()) {
            sum += Character.getNumericValue(bit);
        }

        return sum % tableSize;
    }

    public static void main(String[] args) {
        int key = 10;
        int tableSize = 5;
        System.out.println("Radix Transformation Hash Index: " + hash(key, tableSize));
    }
}
```

Figure 4: Radix Transformation Code

These methods help in efficiently storing and retrieving keys in a hash table by transforming the key values into indices.

Collision Resolution

When two keys hash to the same index (**a collision**), the linked list at that index will contain multiple nodes. Each node represents a key-value pair. When inserting a new key, you check the linked list at the computed index. If the key already exists, update its value; otherwise, append a new node to the list. This ensures that all keys that collide are stored together, maintaining efficient data retrieval. There are different methods on how to handle collisions such as;

1. Open Addressing

All entries are stored within the hash table itself, and collisions are resolved by finding another open slot using probing techniques.

```java
class OpenAddressingHashTable {
    private Integer[] table;

    public OpenAddressingHashTable(int capacity) {
        table = new Integer[capacity];
    }

    private int hash(int key) {
        return key % table.length;
    }

    public void insert(int key) {
        int index = hash(key);
        while (table[index] != null) {
            index = (index + 1) % table.length;
        }
        table[index] = key;
    }

    public void delete(int key) {
        int index = hash(key);
        while (table[index] != null) {
            if (table[index].equals(key)) {
                table[index] = null;
                return;
            }
            index = (index + 1) % table.length;
        }
    }

    public boolean search(int key) {
        int index = hash(key);
        while (table[index] != null) {
            if (table[index].equals(key)) return true;
            index = (index + 1) % table.length;
        }
        return false;
    }
}
```

Figure 4. Open Addressing Code

Linear Probing: If a collision occurs, check the next index sequentially until an empty slot is found.

2. Chaining

Each index in the hash table points to a linked list (or another collection) that stores all entries that hash to that index.

How It Works? When a collision occurs, the new key-value pair is added to the linked list at that index, allowing multiple entries to coexist at the same index.

```
import java.util.LinkedList;

class ChainingHashTable {
    private LinkedList<Integer>[] table;

    public ChainingHashTable(int capacity) {
        table = new LinkedList[capacity];
        for (int i = 0; i < capacity; i++) table[i] = new LinkedList<>();
    }

    private int hash(int key) {
        return key % table.length;
    }

    public void insert(int key) {
        if (!table[hash(key)].contains(key)) {
            table[hash(key)].add(key);
        }
    }

    public void delete(int key) {
        table[hash(key)].remove(Integer.valueOf(key));
    }

    public boolean search(int key) {
        return table[hash(key)].contains(key);
    }
}
```

Figure 5. Chaining Code

3. Bucket Addressing

A variation of chaining where each index in the hash table contains a bucket that can hold multiple entries.

How It Works? When a collision occurs, the new key-value pair is added to the corresponding bucket, which can be implemented using a linked list, tree, or another data structure.

```java
import java.util.ArrayList;
import java.util.LinkedList;

class BucketHashTable {
    private ArrayList<LinkedList<Integer>> buckets;

    public BucketHashTable(int capacity) {
        buckets = new ArrayList<>(capacity);
        for (int i = 0; i < capacity; i++) buckets.add(new LinkedList<>());
    }

    private int hash(int key) {
        return key % buckets.size();
    }

    public void insert(int key) {
        if (!buckets.get(hash(key)).contains(key)) {
            buckets.get(hash(key)).add(key);
        }
    }

    public void delete(int key) {
        buckets.get(hash(key)).remove(Integer.valueOf(key));
    }

    public boolean search(int key) {
        return buckets.get(hash(key)).contains(key);
    }
}
```

Figure 6. Bucket Addressing Code

Deletion

Deletion in hashing can be handled in a few different ways, depending on the collision resolution strategy used in the hash table. For removing a key-value pair. The hash index is calculated, and the linked list at that index is searched. If the key is found, the node is removed from the list. Here are some methods in deletion;

1. Open Addressing

In open addressing, all elements are stored directly in the hash table. When deleting an element, there are a couple of approaches:

- **Mark as Deleted**: Instead of removing the element entirely, you can mark the slot as deleted. This allows the slot to be reused for future insertions, but you need to handle the search process to skip over these marked slots.
- **Rehashing**: After a deletion, you might need to rehash elements to ensure that all elements can still be found. This approach can be more complex and may require resizing the table if the load factor gets too low.

2. Chaining

In chaining, each slot of the hash table contains a linked list (or another structure) of all elements that hash to the same index. Deletion works as follows:

- Simply remove the element from the linked list at the appropriate index. This is straightforward since you can directly access the list and remove the node without affecting other elements.

3. Bucket Addressing

In bucket addressing is similar to chaining but uses a more complex structure (like a dynamic array) for each bucket:

- **Removal from Buckets**: Like chaining, you can directly access the appropriate bucket and remove the element. The deletion operation can vary depending on the underlying structure of the bucket (e.g., linked list, dynamic array).

Perfect Hash Function

A perfect hash function is a hash function that maps a set of keys to a unique index in a hash table without any collisions. This means that for a given set of keys, each key will have a distinct hash value, allowing for constant-time retrieval. Perfect hashing is particularly useful when the set of keys is static and known in advance, as it can be designed to avoid collisions completely. There are different methods of perfect hash function such as;

Cichelli's method is a constructive approach to creating a perfect hash function, particularly effective for static sets of keys. The method involves the following steps:

1. **Assign Weights**: Each key is assigned a unique weight based on its characteristics. The weights can be derived from the characters of the keys or other properties.
2. **Hash Function Construction**:
    - A primary hash function maps the keys to initial indices based on their weights.
    - If a collision occurs (i.e., two keys map to the same index), a secondary hash function is used to adjust the indices.

The secondary function can be a linear probe or a different mechanism that ensures uniqueness.

3. **Table Size**: The size of the hash table is typically chosen to be large enough to accommodate all keys without collisions, often being a prime number or a power of two to improve performance.

4. **Final Hashing**: The final hash function combines the original hash and the adjustments made by the secondary function to ensure a perfect mapping.

You can watch the tutorial in the link below

https://youtu.be/naKfUA42llw?si=BLYx876Scbij51qP

2.2 FHCD Algorithm (Functional Hashing with Collision Detection)

The **FHCD algorithm** is another method for constructing perfect hash functions. It works as follows:

1. **Initial Hash Function**: Start with an initial hash function that maps keys to indices. This function may produce collisions.

2. **Collision Detection**: When a collision is detected, rather than resolving it immediately, the algorithm records the colliding keys.

3. **Rehashing**: For each group of keys that collide, a secondary hash function is created specifically for that group. This secondary function is designed to map those keys to unique indices within a smaller hash table dedicated to that group.

4. **Dynamic Table Construction**: This method allows the creation of smaller hash tables on-the-fly, enabling a perfect mapping for each set of colliding keys without needing a large overall hash table.

5. **Final Structure**: The final structure consists of the original hash table and additional smaller tables for groups of colliding keys, ensuring that each key has a unique index.

   You can watch the tutorial in the link below

   https://youtu.be/HHQ2QP_upGM?si=kTf5p1fVXW47vtBY