Algorithms and Complexity

Homework-4 Report

Implementation and Analysis of Hash Functions

Objective

The primary objective of this assignment is to design and implement custom hash functions tailored to specific use cases and to create a structure capable of performing fundamental operations within hash tables. This involves developing a detailed understanding of hash functions, learning how to manage collisions effectively, and analyzing the performance of the implemented hash table under various conditions.

A secondary aim of the assignment is to test the designed structure with datasets of varying sizes, identify patterns in its performance, and pinpoint potential areas for improvement. This process provides a deeper insight into the nuances of hash table operations and the broader implications of choosing specific algorithms.

Problem Statement

The focus of this assignment is the design and implementation of a hash table with the following critical features:

- **Custom Hash Function:**
  - The hash function is expected to efficiently map keys to appropriate indices within the hash table. Its design should emphasize uniform distribution to prevent clustering and reduce collision rates.
  - A well-distributed hash function ensures optimal utilization of table buckets, balancing load and maintaining high performance.
- **Collision Handling:**
  - "Separate chaining" will be used as the collision management strategy. Each bucket in the hash table is treated as a linked list, which stores all keys that hash to the same index. This approach effectively decouples bucket capacity from table size, enabling scalability.
- **Core Functions:**
  - The hash table should support essential operations that ensure its functionality and usability:

- **insert(K key, V value):** Adds a key-value pair to the table, placing the key in its corresponding bucket.

```java
@Override
public void insert(K key, V value) {
    int index = hash(key);
    Node<K, V> node = table[index];

    if (node == null) {
        size++;
        table[index] = new Node<>(key, value);
    } else {
        size++;
        while (node.next != null) {
            if (node.key.equals(key)) {
                node.value = value;
                return;
            }
            node = node.next;
        }
        if (node.key.equals(key)) {
            node.value = value;
        } else {
            node.next = new Node<>(key, value);
        }
    }
}
```

- **insertWithAnalyze(K key, V value):** Inserts a key-value pair and simultaneously measures performance metrics such as load factor, collision count, search time, and memory usage.

```java
@Override
public void insertWithAnalyze(K key, V value) {
    long startTime = System.nanoTime();
    insert(key, value);
    long endTime = System.nanoTime();

    double insertTime = (endTime - startTime) / 1e9;

    double loadFactor = (double) size / table.length;

    int totalCollisions = 0;
    int bucketCollisions = 0;

    for (int i = 0; i < table.length; i++) {
        int listSize = 0;
        Node<K, V> current = table[i];

        while (current != null) {
            listSize++;
            current = current.next;
        }

        if (listSize > 1) {
            totalCollisions += (listSize - 1);
            if (i == hash(key)) {
                bucketCollisions = listSize - 1;
            }
        }
    }
    int totalMemoryUsage = 0;
    int bucketMemoryUsage = 0;

    for (int i = 0; i < table.length; i++) {
        int tempBucketMemory = 0;
        Node<K, V> temp = table[i];

        while (temp != null) {
            totalMemoryUsage++;
            tempBucketMemory++;
            temp = temp.next;
        }

        if (i == hash(key)) {
            bucketMemoryUsage = tempBucketMemory;
        }
    }

    // hocam collision ve memory usage icin toplam sayi mi yoksa
    // node'un eklendigi bucket'taki toplam sayi mi oldugunu anlamadigim
    // icin iki durumda da sayilarini hesapladim
    System.out.println();
    System.out.println("Total Load Factor: " + loadFactor);
    System.out.println("Total Number of Collisions: " + totalCollisions);
    System.out.println("Total Number of Collisions in Bucket "+ hash(key)+": "+ bucketCollisions);
    System.out.println("Insert Time: " + insertTime + " seconds");
    System.out.println("Total Memory Usage: " + totalMemoryUsage + " nodes");
    System.out.println("Total Memory Usage in Bucket "+ hash(key)+": "+ bucketMemoryUsage + " nodes");
    System.out.println();
}
```

- **search(K key):** Locates and retrieves the value associated with a specified key.

```java
@Override
public V search(K key) {
    int index = hash(key);
    Node<K, V> node = table[index];

    while (node != null) {
        if (node.key.equals(key)) {
            return node.value;
        }
        node = node.next;
    }

    return null;
}
```

- **delete(K key):** Removes the specified key-value pair from the table, ensuring structural integrity.

```java
@Override
public void delete(K key) {
    int index = hash(key);
    Node<K, V> node = table[index];
    Node<K, V> prev = null;

    while (node != null) {
        if (node.key.equals(key)) {
            if (prev == null) {
                table[index] = node.next;
            } else {
                prev.next = node.next;
            }
            size--;
            return;
        }
        prev = node;
        node = node.next;
    }
}
```

- **display():** Visualizes the hash table's complete structure, showing bucket contents and linked list details.

```java
@Override
public void display() {
    for (int i = 0; i < table.length; i++) {
        Node<K, V> node = table[i];
        System.out.print("Bucket " + i + ": ");

        while (node != null) {
            System.out.print("[" + node.key + ": " + node.value + "] -> ");
            node = node.next;
        }
        System.out.println("null");
    }
}
```

- **displayOnlyElements():** Displays only the stored key-value pairs, omitting structural details.

```java
@Override
public void displayOnlyElements() {
    for (Node<K, V> node : table) {
        while (node != null) {
            System.out.println("[" + node.key + ": " + node.value + "]");
            node = node.next;
        }
    }
}
```

Implementation Details

Task 1: Designing the Hash Table

The hash table will be implemented as a Java class. This implementation requires a custom hash function that maps input keys to table indices. The hash function is central to the table's performance and must be designed to ensure uniform distribution, minimizing collisions and maximizing efficiency. This is achieved through careful selection of mathematical algorithms tailored to the properties of the keys.

Hash Function:

- The hash function transforms key values into table indices, using modular arithmetic and other mathematical techniques to achieve uniform distribution.
- A robust hash function reduces the likelihood of clustering, thereby optimizing the table's load factor and minimizing search times.
- The design process involves analyzing potential input patterns to prevent any skewed distribution that could degrade performance.

```
// Olusturmus oldugum hash fonksiyonu, parametre olarak aldigi degeri
// string bir ifadeye cevirip her bir karakterinin ASCII degerinin kupunu
// alip hepsini toplayarak cikan sonucu 13'e boluyor. Daha sonrasinda da
// hash tablosuna gore modunu alarak bize bir indeks degeri uretiyor.
private int hash(K key) {
    String stringKey = key.toString();
    int hashIndex;
    int asciiCubeSum = 0;
    int i;

    for(i = 0;i<stringKey.length();i++){
        asciiCubeSum += ((stringKey.charAt(i)*stringKey.charAt(i)*stringKey.charAt(i)));
    }

    hashIndex = (int)(asciiCubeSum/13) % table.length;

    return hashIndex;
}
```

Task 2: Handling Collisions

In this assignment, collision handling is achieved using "separate chaining." This method ensures that all keys mapping to the same index are stored in a linked list associated with that bucket.

- **Advantages:**
  - Separate chaining decouples the number of keys from the table's fixed size, allowing flexibility in handling large datasets.
  - Even with high load factors, the table remains functional, as linked lists grow dynamically to accommodate new entries.
- **Disadvantages:**
  - As collision rates increase, linked lists become longer, leading to higher search times and potential memory overhead.
  - Balancing list length and table size becomes crucial for maintaining optimal performance.

```
int totalCollisions = 0;
int bucketCollisions = 0;

for (int i = 0; i < table.length; i++) {
    int listSize = 0;
    Node<K, V> current = table[i];

    while (current != null) {
        listSize++;
        current = current.next;
    }

    if (listSize > 1) {
        totalCollisions += (listSize - 1);
        if (i == hash(key)) {
            bucketCollisions = listSize - 1;
        }
    }
}
```

Task 3: Performance Measurements

The insertWithAnalyze method is a critical component of this implementation, collecting and reporting comprehensive performance metrics during key insertion. These metrics include:

1. **Load Factor:**
    - Calculated as the ratio of the number of entries to the table size.
    - A higher load factor indicates a more densely populated table, increasing the risk of collisions.

```
double loadFactor = (double) size / table.length;
```

2. **Number of Collisions:**
    - Tracks how often multiple keys map to the same bucket during insertion.
    - This metric highlights the effectiveness of the hash function and the need for resizing or rehashing.

```
int totalCollisions = 0;
int bucketCollisions = 0;

for (int i = 0; i < table.length; i++) {
    int listSize = 0;
    Node<K, V> current = table[i];

    while (current != null) {
        listSize++;
        current = current.next;
    }

    if (listSize > 1) {
        totalCollisions += (listSize - 1);
        if (i == hash(key)) {
            bucketCollisions = listSize - 1;
        }
    }
}
```

3. **Search Time:**
    - Measures the time taken to locate an inserted key within the table.
    - Efficient implementations maintain consistently low search times across datasets.

```
long startTime = System.nanoTime();
insert(key, value);
long endTime = System.nanoTime();
```

4. **Memory Usage:**
    - Reflects the total memory consumed by linked lists in all buckets.
    - Provides insights into the trade-offs between separate chaining and other collision management methods.

```
int totalMemoryUsage = 0;
int bucketMemoryUsage = 0;

for (int i = 0; i < table.length; i++) {
    int tempBucketMemory = 0;
    Node<K, V> temp = table[i];

    while (temp != null) {
        totalMemoryUsage++;
        tempBucketMemory++;
        temp = temp.next;
    }

    if (i == hash(key)) {
        bucketMemoryUsage = tempBucketMemory;
    }
}
```

Task 4: Performance Testing

The hash table will undergo rigorous testing with datasets of varying sizes to assess its robustness and efficiency:

- **Small Dataset:** 10 keys. Expected to demonstrate negligible collisions and low memory usage.
- **Medium Dataset:** 100 keys. Expected to reveal moderate load factor and manageable collision rates.
- **Large Dataset:** 1000 keys. Expected to highlight challenges such as increased collision rates and memory usage.

Performance metrics will be recorded and analyzed for each dataset, with specific attention paid to how the table scales under load and the trade-offs between memory and time complexity.

Task 5: Results and Observations

The performance analysis will emphasize the effectiveness of the hash function and the collision management strategy. Key observations include:

1. **Small Datasets:**
   - Minimal collisions and near-instantaneous search times are anticipated, showcasing the efficiency of the hash function.
2. **Medium Datasets:**
   - The load factor will increase, leading to a slight rise in search times, but the table should remain performant.
3. **Large Datasets:**
   - Collision rates and memory usage are expected to rise significantly, highlighting areas for optimization in the hash function and resizing strategy.

These insights will inform future improvements and guide the development of more scalable hash table implementations. Due to my computer's performance, I used 6, 12 and 18 keys to test for 10 sized table.

6 Keys	Bucket 0: [watermelon: Fruit: Watermelon] -> null Bucket 1: null Bucket 2: null Bucket 3: [apple: Fruit: Apple] -> null Bucket 4: [cherry: Fruit: Cherry] -> null Bucket 5: [banana: Fruit: Banana] -> null Bucket 6: null Bucket 7: [peach: Fruit: Peach] -> null Bucket 8: null Bucket 9: [melon: Fruit: Melon] -> null	Total Load Factor: 0.6 Total Number of Collisions: 0 Total Number of Collisions in Bucket 5: 0 Insert Time: 3.8E-6 seconds Total Memory Usage: 6 nodes Total Memory Usage in Bucket 5: 1 nodes
12 Keys	Bucket 0: [watermelon: Fruit: Watermelon] -> null Bucket 1: null Bucket 2: [onion: Vegetable: Onion] -> null Bucket 3: [apple: Fruit: Apple] -> [garlic: Vegetable: Garlic] -> null Bucket 4: [cherry: Fruit: Cherry] -> [cucumber: Vegetable: Cucumber] -> null Bucket 5: [tomato: Vegetable: Tomato] -> [banana: Fruit: Banana] -> null Bucket 6: null Bucket 7: [peach: Fruit: Peach] -> null Bucket 8: [pumpkin: Vegetable: Pumpkin] -> null Bucket 9: [melon: Fruit: Melon] -> [potato: Vegetable: Potato] -> null	Total Load Factor: 1.2 Total Number of Collisions: 4 Total Number of Collisions in Bucket 5: 1 Insert Time: 5.6E-6 seconds Total Memory Usage: 12 nodes Total Memory Usage in Bucket 5: 2 nodes
18 Keys	Bucket 0: [watermelon: Fruit: Watermelon] -> null Bucket 1: [water: Drink: Water] -> [coffee: Drink: Coffee] -> null Bucket 2: [onion: Vegetable: Onion] -> [coke: Drink: Coke] -> [lemonade: Drink: Lemonade] -> null Bucket 3: [apple: Fruit: Apple] -> [garlic: Vegetable: Garlic] -> null Bucket 4: [cherry: Fruit: Cherry] -> [cucumber: Vegetable: Cucumber] -> null Bucket 5: [tomato: Vegetable: Tomato] -> [banana: Fruit: Banana] -> null Bucket 6: null Bucket 7: [peach: Fruit: Peach] -> null Bucket 8: [pumpkin: Vegetable: Pumpkin] -> [tea: Drink: Tea] -> [juice: Drink: Juice] -> null Bucket 9: [melon: Fruit: Melon] -> [potato: Vegetable: Potato] -> null	Total Load Factor: 1.8 Total Number of Collisions: 9 Total Number of Collisions in Bucket 5: 1 Insert Time: 5.3E-6 seconds Total Memory Usage: 18 nodes Total Memory Usage in Bucket 5: 2 nodes

- When we increased the number of keys while the table size was fixed, there was an increase in the load factor, collision and memory usage. Also, the probability of the number of collisions and memory usage in any bucket increasing increased.

Final Conclusions

This assignment has provided a comprehensive exploration of the design and implementation of hash functions. The insights gained from collision handling and performance analysis demonstrate how critical these aspects are to the efficiency of hash tables in real-world applications.

To further refine the hash table's performance:

- **Enhanced Hash Functions:**
  - Experiment with more sophisticated algorithms to achieve even better key distribution and minimize collisions.
- **Alternative Collision Handling Methods:**
  - Investigate methods like open addressing, double hashing, or other hybrid approaches.
- **Dynamic Table Resizing:**
  - Implement dynamic resizing strategies to adapt the table size based on load factor thresholds, maintaining optimal performance across varying workloads.

By addressing these areas, the hash table can be made more robust, scalable, and efficient, ensuring its utility in diverse computational scenarios.

Name: Yunus Emre

Surname: Doğan

No.: 26012579136