Lab Assignment #7

HashMap

Full Name: Shreyas Srinivasa
BlazerId: SSRINIVA

CS 303 Algorithms and Data Structures

Spring 2023

1. Problem Specification

The purpose of this project was to implement a hashing function and a hashmap that uses the function. We try linear probing and quadratic probing techniques in our hashmap operations for collision resolution. The performance of the get operation using search keys is evaluated by measuring the time taken to search each implementation of the hashmap, which all contain inserted key value pairs from a sizeable dataset. We use this experimental data to analyze and come to a conclusion on which technique provides us the best performance in this situation

2. Program Design

```
public Long H(Long key) {
    return key % maxSize;
}

public Long hashKey(Long key) {
    Long hashKey = H(key);
    Long initialHashKey = hashKey;

    do {
        if (table[Math.toIntExact(hashKey)] != null) {
            if (table[Math.toIntExact(hashKey)].getKey().equals(key)) {
                return hashKey;
            } else {
                hashKey = H(7 * hashKey + 1);
            }
        } else {
            return hashKey;
        }
    } while (hashKey != initialHashKey);

    return null;
}
```

The functions for calculating the hash code and the array index for the original hashmap implementation suggested by the assignment

```
public String get(Long key) {
    Long hashKey = hashKey(key);
    if (hashKey != null && table[Math.toIntExact(hashKey)] != null)
        return table[Math.toIntExact(hashKey)].getValue();
    else
        return null;
}

public void put(Long key, String value) {
    Long hashKey = hashKey(key);
    if (hashKey != null) {
        table[Math.toIntExact(hashKey)] = new HashEntry(key, value);
    } else {
        System.out.println("HashMap at capacity");
    }
}
```

The put and get operation implementations used for all hashmap operations

```
public Long hashKey(Long key) {
    Long hashKey = H(key);
    Long initialHashKey = hashKey;
    int count = 1;

    do {
        if (table[Math.toIntExact(hashKey)] != null) {
            if (table[Math.toIntExact(hashKey)].getKey().equals(key)) {
                return hashKey;
            } else {
                hashKey = H(initialHashKey + count);
            }
        } else {
            return hashKey;
        }
        count++;
    } while (hashKey != initialHashKey);

    return null;
}
```

The linear probe implementation

```
public Long hashKey(Long key) {
    Long hashKey = H(key);
    Long initialHashKey = hashKey;
    int count = 1;

    do {
        if (table[Math.toIntExact(hashKey)] != null) {
            if (table[Math.toIntExact(hashKey)].getKey().equals(key)) {
                return hashKey;
            } else {
                hashKey = H(initialHashKey + (count * count));
            }
        } else {
            return hashKey;
        }
        count++;
    } while (hashKey != initialHashKey);

    return null;
}
```

The quadratic probe implementation

```java
public static void upcMap(HashMapInterface hashMap) {
    Integer lineCount = 0;
    final Integer totalLines = 177650;
    try (Scanner fileScanner = new Scanner(Paths.get("UPC.csv"))) {
        while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            String elements[] = line.split(",");
            Long key = Long.valueOf(elements[0]);
            String data = elements[1] + ", " + elements[2];
            hashMap.put(key, data);
            lineCount++;
            System.out.format("%f\n", (lineCount * 1.0 / totalLines) * 100f);
        }
        fileScanner.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
    }
}
```

The function which produces a hashmap of the required implementation by inserting entries from the UPC.csv dataset

```java
public static List<Long> getSearchKeys() {
    List<Long> keys = new ArrayList<>();
    try (Scanner fileScanner = new Scanner(Paths.get("input.dat"))) {
        while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            String elements[] = line.split(",");
            Long key = Long.valueOf(elements[0]);
            keys.add(key);
        }
        fileScanner.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
        System.exit(1);
    }
    return keys;
}
```

The function to read the search keys from input.dat

```java
public static void test1() {
    //HashMap hashMap = new HashMap();
    //LinearProbeHashMap hashMap = new LinearProbeHashMap();
    QuadraticProbeHashMap hashMap = new QuadraticProbeHashMap();
    hashMap.put(100L, "Hello");
    hashMap.put(200L, "World");
    hashMap.put(300L, "Three Hundred");
    hashMap.put(300L, "Now Four Hundred");
    hashMap.put(50000L, "50K");
    //hashMap.put(2, "Zero");
    System.out.println(hashMap.get(100L));
    System.out.println(hashMap.get(200L));
    System.out.println(hashMap.get(300L));
    System.out.println(hashMap.get(50000L));
    System.out.println(hashMap.get(50001L));
    System.out.println(hashMap.get(100L));
    System.out.println(hashMap.get(200L));
}
```

```java
public static void test2() {
    HashMapInterface hashMap = new HashMap(1500000);
    //HashMapInterface hashMap = new LinearProbeHashMap();
    //HashMapInterface hashMap = new QuadraticProbeHashMap();
    upcMap(hashMap);

    System.out.println(hashMap.get(28785103105L));
    System.out.println(hashMap.get(28400085168L));
    System.out.println(hashMap.get(26297107420L));
    System.out.println(hashMap.get(15400029186L));
    System.out.println(hashMap.get(11545338367L));

}
```

The functions used to test the operations

```java
public static void driver() {
    Instant start, finish;
    long timeElapsed1, timeElapsed2, timeElapsed3;
    String value;
    List<Long> searchKeys = getSearchKeys();

    HashMapInterface hashMap1 = new HashMap(1500000);
    HashMapInterface hashMap2 = new LinearProbeHashMap();
    HashMapInterface hashMap3 = new QuadraticProbeHashMap();

    upcMap(hashMap1);
    upcMap(hashMap2);
    upcMap(hashMap3);

    for(Long key: searchKeys) {
        System.out.format("Key := %d\n", key);
        start = Instant.now();
        System.out.format("Normal := %s\n", hashMap1.get(key));
        finish = Instant.now();
        timeElapsed1 = Duration.between(start, finish).toNanos();

        start = Instant.now();
        System.out.format("Linear Probe := %s\n", hashMap2.get(key));
        finish = Instant.now();
        timeElapsed2 = Duration.between(start, finish).toNanos();

        start = Instant.now();
        System.out.format("Quadratic Probe := %s\n", hashMap3.get(key));
        finish = Instant.now();
        timeElapsed3 = Duration.between(start, finish).toNanos();

        System.out.println("Time taken := ");
        System.out.format("Normal := %d ns Linear Probe := %d ns Quadratic Probe := %d ns\n", timeElapsed1, timeElapsed2, timeElapsed3);
    }
}
```

The driver function which hands us our desired output

3. **Output**

```
Key := 79
Normal := , INDIANA LOTTO
Linear Probe := , INDIANA LOTTO
Quadratic Probe := , INDIANA LOTTO
Time taken :=
Normal := 248533 ns Linear Probe := 127586 ns Quadratic Probe := 57679 ns
Key := 93
Normal := , treo 700w
Linear Probe := , treo 700w
Quadratic Probe := , treo 700w
Time taken :=
Normal := 41162 ns Linear Probe := 18712 ns Quadratic Probe := 13693 ns
Key := 123
Normal := , Wrsi Riversound cafe cd
Linear Probe := , Wrsi Riversound cafe cd
Quadratic Probe := , Wrsi Riversound cafe cd
Time taken :=
Normal := 37675 ns Linear Probe := 25009 ns Quadratic Probe := 15870 ns
Key := 161
Normal := , Dillons/Kroger Employee Coupon ($1.25 credit)
Linear Probe := , Dillons/Kroger Employee Coupon ($1.25 credit)
Quadratic Probe := , Dillons/Kroger Employee Coupon ($1.25 credit)
Time taken :=
Normal := 15603 ns Linear Probe := 15035 ns Quadratic Probe := 13316 ns
Key := 2140000070
Normal := , Rhinestone Watch
Linear Probe := , Rhinestone Watch
Quadratic Probe := , Rhinestone Watch
Time taken :=
Normal := 49960 ns Linear Probe := 37019 ns Quadratic Probe := 18346 ns
Key := 2140118461
Normal := , """V"": Breakout/The Deception  VHS Tape"
Linear Probe := , """V"": Breakout/The Deception  VHS Tape"
Quadratic Probe := , """V"": Breakout/The Deception  VHS Tape"
Time taken :=
```

Output 1.1 Calculated search times

```
Linear Probe := njhjhn, gjfhjbgkj
Quadratic Probe := njhjhn, gjfhjbgkj
Time taken :=
Normal := 15466 ns Linear Probe := 15079 ns Quadratic Probe := 13438 ns
Key := 2160500567
Normal := 2.25 oz (64)g, Dollar Bar Rich Raspberry
Linear Probe := 2.25 oz (64)g, Dollar Bar Rich Raspberry
Quadratic Probe := 2.25 oz (64)g, Dollar Bar Rich Raspberry
Time taken :=
Normal := 15192 ns Linear Probe := 13183 ns Quadratic Probe := 12675 ns
Key := 2172307284
Normal := , Mixed seasonal flower bouquet
Linear Probe := , Mixed seasonal flower bouquet
Quadratic Probe := , Mixed seasonal flower bouquet
Time taken :=
Normal := 15725 ns Linear Probe := 15318 ns Quadratic Probe := 16222 ns
Key := 2177000074
Normal := , 4 way 13 AMP Extension Lead (Wilkinson UK)
Linear Probe := , 4 way 13 AMP Extension Lead (Wilkinson UK)
Quadratic Probe := , 4 way 13 AMP Extension Lead (Wilkinson UK)
Time taken :=
Normal := 18339 ns Linear Probe := 131902 ns Quadratic Probe := 20843 ns
Key := 2184000098
Normal := 21 oz, Christopher's Assorted Fruit Jellies
Linear Probe := 21 oz, Christopher's Assorted Fruit Jellies
Quadratic Probe := 21 oz, Christopher's Assorted Fruit Jellies
Time taken :=
Normal := 38937 ns Linear Probe := 40167 ns Quadratic Probe := 20473 ns
Key := 2187682888
Normal := , fairway
Linear Probe := , fairway
Quadratic Probe := , fairway
Time taken :=
Normal := 16021 ns Linear Probe := 15196 ns Quadratic Probe := 14102 ns
```

Output 1.2 Calculated search times

4. Analysis and Conclusions

Strengths and Weaknesses of the HashTable by Interview Cake

	Average	Worst Case
space	$O(n)$	$O(n)$
insert	$O(1)$	$O(n)$
lookup	$O(1)$	$O(n)$
delete	$O(1)$	$O(n)$

Strengths:

- **Fast lookups:** Lookups take O(1) time on average.
- **Flexible keys:** Most data types can be used for keys, as long as they're hashable.

Weaknesses:

- **Slow worst-case lookups:** Lookups take O(n) time in the worst case.
- **Unordered:** Keys aren't stored in a special order. If you're looking for the smallest key, the largest key, or all the keys in a range, you'll need to look through every key to find it.
- **Single-directional lookups:** While you can look up the value for a given key in **O(1)** time, looking up the keys for a given value requires looping through the whole dataset—**O(n)** time.
- **Not cache-friendly:** Many hash table implementations use linked lists, which don't put data next to each other in memory.

Time and Space Complexity for Linear Probing by Gabriel from OpenGenusIQ:

For Insertion operation, the complexity analysis is as follows:

- Best Case Time Complexity: **O(1)**
- Worst Case Time Complexity: **O(N)**. This happens when all elements have collided and we need to insert the last element by checking free space one by one.
- Average Case Time Complexity: **O(1)** for good hash function; O(N) for bad hash function

Assuming the hash function uniformly distributes the elements, then the average case time complexity will be constant O(1). In the case where hash function work poorly, then the average case time complexity will degrade to O(N) time complexity.

- Space Complexity: **O(1)** for Insertion operation

For Deletion operation, the complexity analysis is as follows:

- Best Case Time Complexity: **O(1)**
- Worst Case Time Complexity: **O(N)**
- Average Case Time Complexity: **O(1)** for good hash function; **O(N)** for bad hash function
- Space Complexity: **O(1)** for deletion operation

The ideas are similar to insertion operation.

Similarly for Search operation, the complexity analysis is as follows:

- Best Case Time Complexity: **O(1)**
- Worst Case Time Complexity: **O(N)**
- Average Case Time Complexity: **O(1)** for good hash function; **O(N)** for bad hash function
- Space Complexity: O(1) for search operation

Quadratic Probing Insights by Gabriel from OpenGenusIQ:

Advantages of Quadratic Probing

- It is used to resolve collisions in hash tables.
- It is an open addressing scheme in computer programming.
- It is more efficient for a closed hash table.

Disadvantage of Quadratic Probing

- It has secondary clustering. Two keys have the same probe sequence when they hash to the same location.

Compared to other hash methods
In linear probing, when collision occurs, we linearly probe for the next bucket and we keep probing until an empty bucket is found. In quadratic probing, we probe for the $i^{2\,th}$ bucket in i^{th} iteration and we keep probing until an empty bucket is found. In double hashing, we use another hash function $hash_2(x)$ and look for i * $hash_2(x)$ bucket in i^{th} iteration. It requires more computation time as two hash functions need to be computed.

This concludes that linear probing has the best cache performance but suffers from clustering, quadratic probing lies between the two in terms of cache performance and clustering while double caching has poor cache performance but no clustering.

As we already know from previous labs:-
The Time Complexity of an algorithm/code is **not** equal to the actual time required to execute a particular code, but the number of times a statement executes.

The **actual time required to execute code is machine-dependent**.

Instead of measuring actual time required in executing each statement in the code,Time Complexity considers how many times each statement executes.

Observations based on the lab experiment

- The time measurements can be differentiated and compared with each other only when converted to nanoseconds. This clearly shows that the average performance of O(1) for search operations for hashmaps is true. During consecutive insertion operations, the linear probing implementation is the slowest as it requires sequential probing for empty space on collision.
- No clear conclusion can be made from the results. The performance of the hashmap implementation ultimately depends on the quality of the hash function as well as the nature of the dataset, along with other situational restrictions like space.

5. References

- https://www.geeksforgeeks.org/java-program-to-implement-hashtables-with-linear-probing/
- https://www.geeksforgeeks.org/quadratic-probing-in-hashing/
- https://www.interviewcake.com/concept/java/hash-map
- https://iq.opengenus.org/linear-probing/
- https://iq.opengenus.org/quadratic-probing/