Lab 12 Hash Tables

Weichen Chai

Fall 2022

Introduction

In this assignment, we are asked to work with "Hash", which is a data structure that processes keys and values to the hash table. In this assignment, we will implement three versions of the hash table to show their advantages and disadvantages.

Background

Hash table maps keys to values, in order to store the key and values, we can use arrays to store them. As previously tested in other assignments, the time complexity for Binary trees functions of add, delete and search are O(log n), hash table however, have an advantage for storing and searching for data, and the operations mentioned above can all be performed in a average time complexity of O(1).

Tasks

A table of zip codes

In this task, we want to complete the node class, and read the csv formatted files given to us from the assignment. In the file, 9675 Swedish regions with their zip codes are given.

When writing the node

```
//String code changes to Integer code for next task
      public Node(String code, Integer pop, String name) {
            this.code = code;
            this.name = name;
            this.pop = pop;
        }
        ...
```

In this task we are asked to write a lookup method as well as benchmark the searches for "111 15" and "984 99"(Typo in document, should only be 984 99 instead of 994 99). Further more, we compare it to changing the node to hold integer codes instead, and replace a few lines of code to populate the data.

```
while ((line = br.readLine()) != null) {
                String[] row = line.split(",");
                code = Integer.valueOf(row[0].replaceAll("\\s", ""));
                data[i++] = new Node(code, row[1], Integer.valueOf(row[2]));
            }
```

For the linear search, we compare the strings using equals method.

```
    public String Lsearch(Integer code) {
        for (int i = 0; i < data.length; i++) {
            if (code.equals(data[i].code)) {
                return data[i].name;
            }
...
```

The data received after comparison is the following:

Search	**984 99**	**111 15**
Linear	89	85997
Binary	467	478

Table 1: String code table

Search	**984 99**	**111 15**
Linear	106	22597
Binary	226	208

Table 2: Integer code table

The trend of binary search following O(nlogn) and Linear search following O(n) never changed, but the difference in time when using integer and string can be explained through their properties, as integer is quicker than string due to the complex comparison from one string to another. Their names are Stockholm and Pajala respectively, they are not mentioned in the table.

Size matters

In order to make the algorithm more efficient, and since the array we have is 90 percent empty as of now, this means that we should resize the original key into index of smaller arrays using the method of hashing. As shown in the file given, there are 9676 lines. So we begin with resizing the keys to an array size of 9676. This is done in the Zip class used before, however, with a few minor changes. Furthermore, in hashing operation, we perform the following

```
        if (code.equals(p.code)) {
                p = p.next;
                break;
            }
            last = p;
            p = p.next;
        }
        if (last != null) {
            last.next = n;

        } else {
            data[key] = n;
        }
...
```

If we have two key that is mapped to the same index then there will be a collision, the codes for collision was given to us in the task, we also need to find the number of collisions that occur, and this can be done by going through all the keys, and then count the number of collisions.

```
    if (temp.add(keys[i] % mod) == 0) {
        collision++;
```

When running the codes checking for collisions, we get the following table

Size	**Collision** }
10000	5210
12345	2529
20000	3271
22345	1292
30000	2407
32345	769
100000	0

As the table above shows, as more mods are added, the less collisions occur, at an expense of more memory usage, however, it is visible that whole numbers such as 10000 and 20000 mods have much more collision than a sequence of random numbers. This may be due to the fact that modulo is used in this calculation, making whole numbers a lot more likely to receive a collision than other variables.

Handling collisions

slightly better?

In order to execute a more efficient version of the bucket implementation shown above, is to use the array directly by itself. The lookup or search function will stop when it discovers an empty spot.

```
    public String lookup(Integer code) {
...

        Node c = this.data[index];
        while (c != null) {
            if (code.equals(c.code))
                return c.name;
            c = c.next;
        }
...
```

The task asks for us to derive a table which shows the amount of elements to look before finding the correct one. The data shown below are elements traversed to reach an empty space after finding collision.

Size	**Traverse**
20000	37
22345	10
30000	16
32345	8
40000	14
42345	7.5
100000	0