Hash tables

Pontus Hector

Fall 2022

Introduction

This report is conducted as the ninth assignment in the course ID1021. In this report one explores different implementations of storing and accessing larger sets of data and their benefits and drawbacks.

Intuitive construction

In the first task one is supposed to extract information from a given file and store the information as sequential nodes in an array. After this is done, a linear search method and a binary search method is implemented to search for different zip codes stored as strings in individual nodes. The following measurements were obtained through benchmarking.

Iterations	**Target zip**	**Linear search**	**Binary search**
1000000	11115	0.01μs	5.7μs
1000000	45731	25μs	5.2μs
1000000	99499	52μs	11.4μs

Table 1: Cost of searching for different zips

Since the zip codes stored as strings can be translated to actual numeric values we store them as integers instead, hoping for better runtimes. The following measurements were obtained through benchmarking.

Iterations	**Target zip**	**Linear search**	**Binary search**
1000000	11115	0.05μs	61ns
1000000	45731	15μs	34ns
1000000	99499	31μs	238ns

Table 2: Cost of searching for different zips

Storing the zip codes as integers increased the efficiency remarkably. This is since comparing strings require one to compare each token one by one of both strings and in worst case each and every token. Integers, on the other hand, can be directly numerically compared without further complication and this is the main reason for the improvement in run time.

Using key as index

Searching for a certain key using these algorithms becomes quite inefficient for very large data sets. A different approach would be to know where each zip code is stored which would render the time complexity of retrieving a certain zip constant (O(1)). To enable this each zip code is stored at the array position equivalent to the zip code and to retrieve a zip code one simply retrieves the stored data at that position in the data array. The following measurements were obtained through benchmarking.

Iterations	**Target zip**	**Lookup**
1000000	11115	$4ns$
1000000	45731	$4ns$
1000000	99499	$4ns$

Table 3: Cost of searching for different zips

The main issue with this implementation is that its very inefficient when it comes to memory. We allocate memory for 100000 elements to fit all the different values zip codes can be, in which we use less than 10000 of these memory slots. To work around this we implement a hash function in which we use a smaller sized array where the zip codes are stored at the index equivalent to a predetermined modulo of each and every zip code. The problem with this is that multiple zip codes may obtain equivalent indexes to one another which leads to collisions between zip codes contending about the same index.

Mod\Col	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
10000	4465	2414	1285	740	406	203	104	48	9
12345	7145	2149	345	34	1	0	0	0	0
20000	6404	2223	753	244	50	0	0	0	0
54321	9332	342	0	0	0	0	0	0	0

Table 4: Number of collisions for hashing the given zips

To manage the collisions we implement arrays within the data array which is commonly known as **buckets**. A bucket is created when the first node is inserted at a given index and expanded by 1 memory slot if an additional node is inserted.

```java
public Zip(String file, int mod) {
        ...
        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
            ...
            while ((line = br.readLine()) != null) {
                String[] row = line.split(",");
                Node extracted = new Node(Integer.valueOf(row[0].replaceAll("\\s", "")),
                row[1], Integer.valueOf(row[2]));
                keys[i] = extracted.code;
                key = keys[i] % mod;

                if(data[key] == null){
                    data[key] = new Node[1];
                    data[key][0] = extracted;
                } else {
                    Node[] expand = new Node[data[key].length + 1];
                    for (int k = 0; k < data[key].length; k++){
                        expand[k] = data[key][k];
                    }
                    expand[data[key].length] = extracted;
                    data[key] = expand;
                }
                i++;
            }
        }
        ...
```

This is a viable option but it may be even more refined. Instead of implementing expanding buckets we could store the items directly in the array. The zips will be stored at the predetermined modulo index of that zip to the best of our ability. If the array index is already occupied, the item will be inserted at the first sequentially available index. The lookup method as well need to be implemented in a new way. Instead of searching through the array at a certain index the lookup method should start with looking at the zip code at the predetermined modulo index of that zip. If the zip stored at that index is incorrect the lookup method should evaluate the next index continuously up until the correct zip has been found or until a null node has been reached.

```java
public int lookup(Integer code, int mod){
        int looks = 0;
        int index = code % mod;
        for(int i = index; i < mod; i++){
            looks++;
            if(data[i] == null)
                break;
            else if(data[i].code.equals(code))
                break;
        }
        return looks;
    }
```

The following measurements were obtained through benchmarking.

Modulo	Lookups	Average Lookups
10000	1\6\1501	592
11000	1\29\383	107
12000	1\1\623	92
13000	1\7\15	69
14000	1\15\200	54
15000	1\1\12	53

Table 5: Number of lookups for zips 11115\45731\98499 and average for all zips

The total number of lookups are considerably larger than of the previous implementetion. In the previous implementation there was at most 9 collisions using modulo 10000 which means that the lookup method **at most** needed to iterate 9 times through a bucket to find the correct value. However this implementation is slightly more efficient on utilizing memory and when it comes down to which one is the better choice, its all about a trade-off between utilization of memory and performance.