Assignment 10 Report - Hash tables in Java

Dean Tsankov

October 20, 2024

Introduction

The purpose of this report is to present a data storage methodology, which organizes elements in an easily accessible way given a key value. Known as a Hash table this concept could have a multitude of implementations. Here I will go over some simple yet not so efficient ones, some very efficient time-wise yet not so much space-wise and by the end hopefully reach a stable middle ground solution where both execution time on requesting an element as well as storage utilization are both well optimized.

The work with the hash tables will be carried out over a provided data set of Swedish zip-codes. Each CSV entry consists of a name for the area, the zip code and the population.

Linear and Binary Look-ups

We are provided a skeleton code which reads the data set and enters each element in an array indexed consecutively. The first search implementations we are asked to perform and assess are a linear search through the elements which would of course have an `O(n)` time complexity and a binary search with `O(logn)` time. As both of these have been discussed in previous reports, for the sake of not cramming this one I will skip their implementation and simply present some arbitrary benchmark results we were asked to evaluate.

Postcode	**”111 15”**	**”984 99”**
Linear lookup	135	132482
Binary lookup	7411	850

Table 1: Benchmark on the lookups, output in nanoseconds

As expected, when considering the dataset is ordered based on the zip code value the linear search has low execution time for elements closer the the beginning, but is bad for ones further back the list. On the other hand the binary search has a more general execution time. Note the results here and each table thereafter are average measurements from 100 executions.

Key type optimization

Next working with the same two lookup methods, we try to encode the zip codes not as strings but as cast integer values. This is with consideration to the fact that on look up we always compare the zip codes and it is for certain that the string comparison has to go through each character in order to yield a result while working with numbers equality checks are performed instantly.

Here is how we populate the array now:

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

Bellow is a table with the improvement included:

Postcode	**”111 15”**	**”984 99”**
Linear lookup	657	2993285
Binary lookup	1149	486

Table 2: Benchmark on the lookups, output in nanoseconds

The results here are somewhat contradictory as I understand them. On the one hand, the binary search has had a good improvement, but the linear one seems to perform even worse. There should not be a particular reason for this happening.

Array size discussion

A different approach to have low element get times and in fact as low as they could possible go (i.e. `O(1)`) is to have the key or in this case the zip code be the index for the data entry. The drawback here is that the size of the underlying array needs to be as large as there are possible key values.

In the particular setting we know we have about 10 000 zip codes in the set, but the possible codes themselves reach 100 000, which would mean that our data utilization would be around 10% which while not terrible is not optimal. Even still here is the great linear execution for comparison purposes:

Postcode	**"111 15"**	**"984 99"**
Lookup	1134	1137

Table 3: Benchmark on the lookup, output in nanoseconds

And the simple code addition that makes it work:

```
(...)
while ((line = br.readLine()) != null && i < this.max) {
    String[] row = line.split(",");
    postnr[Integer.valueOf(row[0].replaceAll("\\s",""))] = new
            Area(row[0], row[1], Integer.valueOf(row[2]));
}
(...)
```

Enter Hash functions

The solution to having near linear lookup times and yet reasonably small array size is to continue using the zip code as a key and an index to the data, but somehow transform it in a way. This way is known as a "Hash" function. We would like a hash function to uniquely (as much as possible) map each zip code to a smaller value, which we then use as the index to store in a smaller array. There is definitely a lot of thinking involved in coming up with an optimal hash function, but here we will be content at using the key modulo some constant (related in a way to the size of the array), since modulo gives cyclic outputs. Moreover exactly due to this cyclic output nature the indexes returned from the function will not be completely unique and so we will be writing to the same index multiple times. This is known as a collision. Our task in general is to minimize these as well as to find ways to deal with them without sacrificing data. For this section I will show an exploration of trying to minimize them by using different modulo constants.

The table bellow is an execution of a semi working provided function (some additions were needed) where the first column represents the modulo constant and the rest of the columns show how many single, double, triple and so on collisions occurred on a given element in total.

10000:	2050	1130	545	334	203	99	56	39	9
20000:	4180	1471	509	194	50	0	0	0	0
12345:	4997	1804	311	33	1	0	0	0	0
17389:	6352	1414	161	3	0	0	0	0	0
13513:	5410	1678	287	12	0	0	0	0	0
13600:	3406	1578	613	229	56	13	0	0	0
14000:	3055	1316	615	320	134	31	1	0	0

This gives us good intuition about what our modulo constant should be since we are more content with having single collisions then any other type. We can notice that these outcomes happen when the constant is a prime number, and this would make sense, considering that dividing with a prime number would maximize the amount of different results for the index.

Hash Table Collision Handling

Finally, let us take a look at some ways to handle the inevitable collisions.

Buckets

One approach is to keep a "bucket" ,so to speak, along with every entry in the array. When we add an element at an index where there already happens to be one we instead add it to the respective bucket for that slot. This bucket could be implemented as a linked list for example. Later when we are searching for one of the elements at that index since evidently all of their zip codes would point to the same index, through the hash function, we go through each of the elements in the bucket until we get the one we are searching for. While not constant time complexity, this look up still does not approach linear behaviour and so is very good.

Here is the population routine now:

```
(...)
while ((line = br.readLine()) != null && i < this.max) {
    String[] row = line.split(",");
    int hashedindex = HashFunc(Integer.valueOf(row[0]
        .replaceAll("\\s","")));
    if(postnr[hashedindex] == null){
        postnr[hashedindex] = new Area(row[0], row[1],
            Integer.valueOf(row[2]));
    } else {
```

```
            if (buckets[hashedindex] == null) {
                buckets[hashedindex] = new Bucket();
            }
            buckets[hashedindex].add(new Area(row[0], row[1],
                Integer.valueOf(row[2])));
        }
    }
    (...)
```

And the accompanying lookup function:

```
    (...)
    public Area Lookup(String zip) {
        int hashedZip = HashFunc(Integer.valueOf(zip.replaceAll("\\s", "")));
        if (postnr[hashedZip] != null) {
            if(postnr[hashedZip].zipcode.equals(zip)){
                return postnr[hashedZip];
            } else if(buckets[hashedZip] != null) {
                return buckets[hashedZip].find(zip);
            }
        }
        return null;
    }
    (...)
```

Inline store

Another way we can deal with collisions is the following. When there is an element at the same index we just continue to the right of it until we find an empty spot. Next when we are searching for it if it is not in the hashed index then we would know its somewhere to the right and so we perform somewhat of a linear search over a smaller data set. This size is very much determined by how packed our array is, since it might be that we have to traverse almost all of it in order to find a slot. For this reason an optimal size of the array itself is twice the size of the data, then clustering would be minimized and execution would be fairly good. Removing an element in this solution, though, would require some specific mechanisms outside the scope of this report, but is worthwhile considering.

Below is how we write to the array using this approach:

```
    (...)
    while ((line = br.readLine()) != null && i < this.max) {
        String[] row = line.split(",");
```

```
        int hashedindex = HashFunc(Integer.valueOf(row[0]
            .replaceAll("\\s","")));
        if (postnr[hashedindex] != null) {
            while (postnr[hashedindex] != null) {
                hashedindex++;
                if (hashedindex == this.max) {
                    hashedindex = 0;
                }
            }
        }
        postnr[hashedindex] = new Area(row[0], row[1],
            Integer.valueOf(row[2]));
    }
    (...)
```

And then the fairly simple in concept lookup:

```
    (...)
    public Area Lookup(String zip) {
        int hashedZip = HashFunc(Integer
            .valueOf(zip.replaceAll("\\s", "")));
        while (postnr[hashedZip] != null) {
            if (!postnr[hashedZip].zipcode.equals(zip)) {
                break;
            }
            hashedZip++;
            if (hashedZip == this.max - 1) {
                hashedZip = 0;
            }
        }
        return postnr[hashedZip];
    }
    (...)
```

Bellow, lastly, I present a comparison between the execution times for the lookup operations on the two collision handling solutions.

Code	**Bucket (ns)**	**Inline (ns)**
”111 15”	1480	627
”984 99”	1302	525
”213 15”	1159	613
”964 89”	966	521

Table 4: Average execution on couple of random zipcodes

The array inline store appears to be doing pretty good, and it was the one of the two which was way simpler to implement, but in general seems to have more supporting considerations and hidden problems. The bucket solution, I would presume requires more time due to using a linked data structure as explored in the respective report, when comparing it with an array. Even then in spite of being harder to implement, seems more robust and better scale-able.

Closing

This report required a lot of theoretical explanation and as such would have expanded drastically if I had included all of the code and versions of the classes I wrote for it. I feel that would have made it hard to understand. The code I wrote for this assignment will be uploaded to its respective Github repository for this course, in case it is necessary for understanding how any of the mentioned concepts where implemented. Link