Hash Tables

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to introduce a new way of storing data sets under a comprised form, by making use of hash functions. In this experiment, a data base of Sweden's zip codes has been used as data source, first in order to benchmark searching algorithms, then to compare looking for an Integer in contrast to a String data type and in the end, to hash the zip code keys.

Looking for the first and last zip code in the zip code data base

Linear and Binary search

It should be taken into consideration that, the first zip code in the data base is "111 15" corresponding to Stockholm, while the last one is "984 99" corresponding to Pajala. Both linear and binary search have been implemented in order to look for the two zip codes, which were first compared under the form of an Integer and then, under the form of a String. The search methods in both cases (zip code as Integer and zip code as String) are similar in terms of approach - the only thing that matters is the type of the parameter.

Note: Since we are working with Integers and Strings (objects) in the methods implementation we cannot use '==' in order to compare two values. Instead, the method *compareTo()* or *equals()* should be used.

Zip Code	Type	LinearSearch[ns]	BinarySearch[ns]
111 15	String	23	128
111 15	Integer	23	40
984 99	String	40K	119
984 99	Integer	17K	43

Table 1: Benchmark analysis of different ways of searching for the zip codes.

As a result, the following conclusions can be drawn after having analysed the benchmark:

- binary search undoubtedly *outperforms* the linear search (in spite of the fact that as far as the first zip code is concerned, time is a bit lost using this algorithm - nevertheless, it was something to be expected);
- searching for a String is much *more expensive timewise* compared to looking for an Integer (this can be seen especially for the zip code "984 99" since it is the data base's last element - so the time differences add up until the end, unlike for the zip code "111 15" which is the first element and most probably there is a difference in the time spent to search, however it is too little to be noticeable);

Binary search and direct accessing in an array by index

Before starting the analysis, my assumption was that accessing by index in an array of 100000 elements would be much quicker compared to binary search since it is an O(1) operation. The following *lookup (String zip)* method has been written in order to find a key in such an array:

```
public boolean lookup(String zip) {
    //converting from String to Integer
    Integer c = Integer.valueOf(zip.replaceAll("\\s",""));
    if(data[c].code.equals(c)) //checking if the key is found
        return true;
    else
        return false;
}
```

Zip Code	Type	BinarySearch[ns]	KeyAsIndex[ns]
111 15	Integer	40	230
984 99	Integer	43	234

Table 2: Benchmark analysis between binary search and indexing by key in a large array.

After having run the benchmark and taken a look at the results, I was quite surprise to see that apart from having a 90% empty array, the execution times were on average 5 times higher than the ones resulting by using the binary search. And that is precisely because in my time measurement, I also take into account the time spent converting the parameter from String to Integer.

It should also be taken into consideration that the reported values in the tables above are the result of ***running the search operations 100000 times*** and taking the minimum resulting execution time. In addition, caches have also been warmed up right before the time measurement has been started.

Hashing

In order to fully understand what problems could occur while hashing, the following experiment has been done in order to count the number of collisions by varying the number the modulo operation is executed with. The following have been obtained:

m	0 colls	1 colls	2 colls	3 colls	4 colls	5 colls	6 colls	7 colls	8 colls
10000	4465	2415	1285	740	406	203	104	48	9
20000	6404	2224	753	244	50	0	0	0	0
12345	7146	2149	345	34	1	0	0	0	0
13513	7387	1977	299	12	0	0	0	0	0
13600	5895	2489	911	298	69	13	0	0	0
14000	5472	2417	1101	486	166	32	1	0	0

Table 3: Benchmark analysis of the number of collisions.

As far as I am concerned, the goal is to obtain as spread and balanced number of collisions as possible. Having said that, by looking at the table and by doing 'modulo 10000' we obtain the most equally spread number of collisions. By doing 'modulo 13600' we get the second most one only collision.

Note: No element in my benchmark had 9 collisions, therefore I no longer included that column in the table above.

The Bucket Implementation

In order to hash the zip codes in the data base, the first method used in order to so, was by using buckets. I declared an array of size 31 (I chose a small number randomly) and indexed all zip codes by doing 'modulo 31'. If a collision happens, a linked list pointing from the same index is started and elements are added there. (I added to each node has a 'next' reference as well). My approach consists of **3 methods**:

- one method that returns the value of the hash function;

```
public Integer hashvalue(Integer n) {
    return n % 31;
}
```

- one method that hashes all data base values;

```
public void Hash() {
    for(int i = 0; i <= max; i++) {
        //getting the value of the hash function
        Integer value = hashvalue(data[i].code);
        if(hash[value].code == -1) //check if the spot in the array is empty
            hash[value] = data[i];
        else {
            Node index = hash[value]; //otherwise we initailize a pointer
            //walk through the linked list until its end
            while(index.next != null)
                index = index.next;
            Node p = data[i];
            index.next = p; //insert the new node
            p.next = null; //mark the new end of the list
        }
    }
}
```

- one method that looks for a specific zip code in the hashed buckets;

```
public boolean lookup(Integer zip) {
    int value = hashvalue(zip); //get the value of the hash function
    if(hash[value].code.equals(zip)) //check if it is stored in the array
        return true;
    else { //otherwise look in the linked list starting at that index
          Node index = hash[value].next;
          while(index != null) { //go until the end of the list in order to check
            if(index.code.equals(zip))
                return true;
            index = index.next;
           }
        }
    return false;
}
```

The Clustering Implementation

As far as the clustering implementation is concerned, the approach follows the same structure as the one presented above, however with slight changes:

- no linked lists are created in case of collisions - instead we go to the right as much as possible in order to find an empty spot;
- once the initial array is full, we resize it to a new one of double length (time is lost by copying elements from one array to the other);
- the stop condition in the *lookup* method is the moment we find an empty spot (marked by the value -1) - therefore we can conclude that the element couldn't be found;

Benchmarking the two implementations

Zip Code	BucketImplementation[ns]	ClusteringImplementation[ns]
11115	28	29
43446	600	10K
98499	1.3K	22K

Table 4: Benchmark analysis between bucket and clustering hashing.

As expected, for an element (11115) that can be found in the array from the first try, both operations are equally fast since their time complexity is (O(1)). However, when it comes to looking for elements which can't instantly be identified, the bucket implementation is much more efficient timewise since there is no copying involved in order to resize the data structure and the entire array doesn't have to be traveled through until the first empty spot, so that a conclusion on the presence of an element can be drawn.

Zip Code	BucketImplementation [elem]	ClusteringImplementation[elem]
43446	136	4403
98499	303	9663

Table 5: Benchmark analysis of the number of elements that have to be checked before finding the necessary zip code.

Table 5 shows the number of elements that have to checked before finding the necessary one. From a relevance standpoint, I chose those included zip codes so that they can't be found directly (by one try only). By looking at the numbers, the bucket implementation is by far more efficient since less numbers have to be looked at.

Conclusion

All in all, I personally enjoyed more the Bucket implementation of the hash tables since apart from being more efficient, I think the idea of having a linked list that starts from each spot in the array is much more interesting.