Hash Tables

Emanuel Paraschiv

Fall 2024

Introduction

In this assignment we will try to implement hash tables from scratch, understand their logic, learn how to deal with collisions and even test different implementations.

Zip codes

Given a list of Zip Codes, with the corresponding location and people, we want a way to access this data in the fastest way possible.

First we will try to use the simplest way, a method to do the linear searching called `lookup()`. For performance reasons, we will also compare it with a `binarySearch()` algorithm.

For lookup we get a string and we just iterate in the list until we find the item. If it is not found, we just return `null`:

```
for (int i= 0; i < postnr.length; i++)
    if (postnr[i] != null)
        if (postnr[i].zipC.equals(zipC))
        return postnr[i];
return null;
```

The process for binary search is the same as in the previous assignment. We use the middle pointer to track the item. If we find the it we return it and if we get an traverse it completely and don't find it we return `null`.

To find it we need to take the recursive case into consideration. We move left or right depending if the value is bigger or smaller and call the function again with the new arguments.

If we try to benchmark these 2 methods for 2 different zip codes(the first one, 11115, and the last one, 98499) the results work as expected. The lookup method exhibits inferior performance as it has a linear time complexity while binary search finds the key faster:

Zip Code	Array Size	Time (μs)
11115	10000	500
98499	10000	1300

Table 1: Lookup for String

Zip Code	Array Size	Time (μs)
11115	10000	9.5
98499	10000	13

Table 2: Binary Search for String

However, since the zip codes that we are working with are already in integer form, than we can simply use them as `Integers`. If we do this little change and run our benchmark again we get the following results:

Zip Code	Array Size	Avg Time μs
11115	10000	0.3
98499	10000	160

Table 3: Lookup for Int

Zip Code	Array Size	Avg Time ns
11115	10000	1.5
98499	10000	2

Table 4: Binary Search for Int

One explanation for this can be that Integers are simpler to process, while Strings may take additional time as they have to be converted to ASCII code and then evaluated. This of course takes more time so our searching methods perform better with Integers.

Use index

One intelligent idea is to use the Zip code as a key, like a position in the array. If the last zip code is 984 99, then we can make an array of 100,000 elements and use every zip code as a position.

Thus, we can make the access very simple. Instead of using `lookup()` to iterate until it finds each number, now it can access it directly by accessing the zip code and returning it in one line:

```
return postnr[zipC];
```

Now the `lookup()` method has a constant time complexity, and we can forget about using `binarySearch()`. To prove this, we can try to access both previous zip code and the time will be constant:

Zip Code	Time ns
11115	30
98499	30

Table 5: Lookup execution time

Hashing

Even though the access time has a linear complexity now, we have another problem. We have an array of 100,000 elements but it is 90% empty.

One way to solve it is to transform those zip codes to some smaller numbers that can refer to entries in a smaller array (let's say 10,000 elements).

We can create a function that will create other indices that would fit the smaller capacity of the array. This is called a `hash function` and given a Zip Code it will return a smaller number (called 'index') by using the modulo % operator. This function will be given a Zip Code and it will return an index from 0 to let's say 10,000.

There are some instances when two different Zip Codes will end up having the same key. This situation is called a `collision` because they will both try to access the same slot in memory.

For example, we can try to calculate entries mod 10,000/ 20,000/ 50,000 and get the following values:

Mod	0	1	2	3	4	5	6	7	8
10000	2050	1130	545	334	203	99	56	39	9
20000	4180	1471	509	194	50	0	0	0	0
50000	7309	1183	0	0	0	0	0	0	0

Table 6: Collision Cases

However if we take % with respect to some random numbers, the following results appear, which seem overall better:

Mod	0	1	2	3	4	5	6	7	8
14768	5737	1589	244	7	0	0	0	0	0
28474	7895	854	24	0	0	0	0	0	0
83746	9605	35	0	0	0	0	0	0	0

Table 7: Collision Cases

Anyway, if we try to increase the size of the array, then we won't decrease the number of collisions. I tried as given, the array sizes of 13513, 13600 and 14000 and for all i got the same result (for mod values in the past task).

Case	0	1	2	3
14768	5737	1589	244	7
28474	7895	854	24	0
83746	9605	35	0	0

Table 8: Collision Cases

This table is the same for all 3 array sizes, so I present it only once. An explanation for this may be the fact that the array is already big enough in our case, and increasing it further will lead to no better result.

Buckets

In order to deal with these collision we can use an array of `Buckets`.

We are basically starting from an array of consisting of buckets, each of the buckets storing elements that have the same key(collisions). If the bucket gets full, we dynamically resize it.

First we get the index from the hash function (which only does the mod operation) and check if the buckets at that position are empty (this is how we start). If they are, then we allocate the minimal space:

```
int ind= hashF(zipCode);
if (buckets[ind]== null)
        buckets[ind]= new Bucket();
```

Of course, if there area many collisions then we should resize the buckets. To do this we double the size and copy all elements:

```
// resizing
Bucket buck= buckets[ind];
if (buck.count == buck.areas.length)
    int nsize=buck.areas.length * 2;
    Area[] newA= new Area[nsize];
    for (int i= 0; i < buck.areas.length; i++)
        newA[i]= buck.areas[i];
    buck.areas= newA;
```

The lookup will have the following way of functioning: It will get the zip code using the hash function and try to access that bucket. Then it will simply check each element in that bucket and compare it with the key to be found. If it is not found, we return null.

When we try to benchmark this new implementation, we get the following results for different array sizes:

Size	Avg Lookup Time μs
10000	0.3
15000	0.08
20000	0.06

Table 9: Lookup Times for Code 11115

Using one array

Another idea is to use only one array. If before we had to create new buckets to add elements when we had collisions, now we will make do with only one array that is provided in the beginning. The way of functioning in the following: As before, we start with getting the index from the hash function. Then we check and if it is occupied already we try to go to the next slots until we find a free one to add our element:

```
while (table[ind] != null)  // if occupied go until free slot
    ind = (ind + 1) % size; // check next
table[ind] = area;
```

Same is true for `lookup()`. We get to the index where it should and start comparing the key with the current value. We will increment the index until we find the desired item or until we hit a null value.

This will not perform well however in a small array. If the array is small then there will be a lot of collisions and so elements will be added far away from the hashed index. This will increase the time for both adding a new element and also for searching one already existent (both methods will have to iterate more slots).

This way, instead of being constant, the lookup will tend to be linear if the array is full.

Size	Lookup Time μs	iterations
10000	1.5	18
15000	0.4	4
20000	0.3	5

Table 10: Lookup Times for Zip Code 11115

As we can see, when we try to lookup for the same zip code '11115' it takes us more than than before. Even though the time efficiency has decreased, the memory efficiency is better, so there is a trade-off.