Hash tables in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment, I study a new data structure called a hash table, that organizes a set of entries and has them accessible given a key. To see how they can be implemented and utilised, I look at a table of Swedish zip codes. My data comes from the `postnummer.csv` file that was provided to us by the course. Each of its entries consist of the zip code, the name of the area and the population.

Basic implementation

In our first initialisation, we have an array of Areas, so all we do is read the file, store each entry in an Area object and store the reference to it in the array. Now for the lookup, I first use a simple linear search, then binary search and compare them in a benchmark. I use `.equalsTo()` instead of `==` in both of them, because I want to compare the values, not their addresses.

searching for:	”111 15”	”984 99”
linear search	0.1 µs	73 µs
binary search	0.5 µs	0.5 µs

Searching for different zip codes as strings

Since our data is ordered, searching for the first element in the array is a lot faster than searching for the last one in the array, because we don’t iterate through the entire array. As expected, the binary search has execution time independent of the position of the key in the array, so it was far better for looking up ”984 99” which is the last zip code, and it did not significantly worsen looking up the first element either.

zip codes as integers

Next thing we notice, is that all the zip codes are integers, so we can cast them before storing them in the Area object. The same benchmark but with

the keys being integers this time yields the following results:

searching for:	11115	98499
linear search	<0.1 µs	15 µs
binary search	0.2 µs	0.2 µs

Searching for different zip codes as integers

As you can see, all four execution times improved. That is due to the fact the Strings are not primitive data types, so the `.equals()` comparison has to iterate over the string character by character (`==` would be quicker, but not always reliable).

Improved implementation

We can further improve execution times if we use an array that has the keys as its indices. To be able to do this, we increase the size of the array to 100.000 (as there are 99.999 possibilities for the Swedish zip codes) and change the Area object so that it only stores a name and population, and uses the zip code as an index when populating the array.

binary search before	lookup with the new implementation
0.2	<0.1
0.2	<0.1

This implementation is definitely the most time-efficient, however, it has a drawback too. There are only 9675 entries, yet we require an array of size 100.000 to handle them, so about 90% of the array is empty. The solution to this is to use something known as a hash function, that transforms a key to an index in a smaller array.

defining a hash function

A simple hash function would be taking the modulo m of the key, with suitable value for m. With such function, it will more than likely happen that two different keys will be hashed to the same index, so a collision occurs. Later we will implement a technique for handling these collisions, but it will have a time penalty, so for now, the goal is to try and minimise the amount of collisions by choosing our m strategically, so that the indices end up being fairly unique. I set up my test to measure the number of elements hashed to each index position for different values of m:

m	only 1 value	2	3	4	5 or 5+
10000	2050	1130	545	334	406
15000	3171	1267	625	296	178
20000	4180	1471	509	194	50
10007	3656	1713	599	153	36
12345	4997	1804	311	33	1
17389	6352	1414	161	3	0
20021	6114	1434	215	12	0
20323	7201	1125	72	2	0

As you can see, there is some kind of relationship between size and the number of collisions, but also the more "random" numbers tend to perform better. To be more precise, primes are some of the best choices for m, as real-life data tend to have some periodicity that we do not want to pick up. Moving forward, I will be using m=17389.

Handling collisions

One way to handle collisions is to store references to "buckets" of Area object in the array (instead of the Area objects themselves). This way, if the hash function returns an index to a position that is already non-null, we can just follow the reference and allocate memory there for one more element. I also had to modify my Area object to store the codes again, so that in the lookup once we found the correct bucket, we can determine if the element we're looking for is even in there.

An even better implementation

We can still try and use even less memory by implementing the same idea (with the buckets) but in place. If the position we get from the hash function is already full, we move to the right until we find an empty position. This also changes the lookup, since we need to check not only the index from the hash function, but also every value to the right until there is an empty space. Now if we were to add a remove method to our data structure, we would also have to think about adding a special node when removing, so that the lookup method is still aware that the value may be further to the right.
For my choice of m, there were enough space for all elements, but if this wasn't the case, I could've maybe implemented a similar wrap-around feature as in the D assignment, or just lengthen the array with choosing a different m.

tests

To test how well this performs, I changed the lookup function to return an integer value expressing how many elements we have looked at before finding the correct zip code. I went through the entire list of zip codes and calculated the average and the maximum number of elements that the lookup function traversed (not the minimum, because that is just 1). I found that on average we only looked at 3 elements before finding the one, while the max was 244. I also compare these for different values of m:

max size (m)	average	maximum
10007	7260	567
12345	658	12
17389	244	3
20021	394	4
20323	154	2

Entries that has no empty spaces after their calculated indices (e.g. 37033 and 37034 for m=12345) are not added to array but are in the list of zip codes. As we know, looking up keys that are not in the data is usually the worst case scenario, so I also counted how often that occurs with each different m:

10007	12345	17389	20021	20323
162	2	0	0	0

Summary

In this assignment I compared different values for the number m that can be used for modular hashing. The following table contains a summary of my results for 9675 entries comparing the number of collisions, the number of entries that didn't get stored, and the average and maximum number of elements traversed in a lookup.

m	non-collisions[1]	entries missed	average	max
10007	3656	162	7260	567
12345	4997	2	658	12
17389	6352	0	244	3
20021	6114	0	394	4
20323	7201	0	154	2

All the code I wrote will be uploaded to my github profile.

[1] nr of elements that have a unique hash value