Trees - Breadth-first Search

Daniel Sherif

Fall 2024

Introduction

In the following report, we will explore hash tables and compare the hash tables to some objectively simpler solutions. The comparisons will eventually lead us to some conclusions which helps derive clear pros and cons out of hash tables and learn about their benefits.

Setup

We will start by setting up our table. We are going to work with a generally undisclosed (for this report) CSV file filled with a bunch of Swedish zip codes, from that testing different methods. The general setup will be ignored as it is not of much importance, putting more focus on the methods themselves to help conclude benefits and explore hash tables.

Lookup in a Table - Strings Versus Integers

In this section, two lookup methods will be tested and reflected upon. First one where Strings will be searched for and the second searching for Integers. The methods will be a slower one and a faster one, a linear search, which goes through the whole array, and a binary one, which basically searches in blocks and focuses that block iteratively depending on results of a sorted array compared to a assigned key. The key we will search for is one which is at the start of the table and one which is at the end, table referencing the array derived of the CSV file. For further discussions sake, we will disclose the keys used. First one is ”111 15” and the last one is ”984 99”.

Following tables, Table 1 and 2, will disclose the results when comparing strings. Table 1 tests the number ”111 15” and Table 2 the number ”984 99”. We expect ”111 15” to take longer in binary but faster in linear and the opposite for ”984 99”, because it is very deep in the table and the linear search needs to go through every element. Note that the times are approximate min-times and cannot give exact values.

Type	**Linear**	**Binary**
String	100 ns	1000 ns
Integer	0 ns	400 ns

Table 1:
Min times received for "111 15", testing linear and binary search methods. Strings and Integers tested.

Type	**Linear**	**Binary**
String	38 μs	1.7 μs
Integer	12.8 μs	1.5 μs

Table 2:
Min times received for "985 99", testing linear and binary search methods. Strings and Integers tested.

As seen in Table 1, when testing for the first number of the table, the linear search is almost instant, both for Strings and Integers, maybe more so for Integers, but this is negligible. For binary search, we get more than double the efficiency when comparing integers instead of Strings. As seen for the last number of the table, Table 2, the linear search is much more efficient when testing linear. Much less so when testing binary. A fair conclusion would be that for binary search, the change in times is less, but still there, while for linear, it is a much bigger difference between Integers and Strings.

This is most probably due to the time it takes for comparisons. More numbers to go through with strings in contrast to integers, which only have the actual integers tested, the zip codes in this case. As a binary search uses much less direct comparisons then linear, the results make a lot of sense.

It is important to note the usage of `.compareTo(key)` or `.equals(key)` instead of simply using two equal signs. If you were to use equal sign to compare, for strings you would not be comparing the strings themselves, rather the reference of the string, the place its stored. While for integers same applies, for larger integers, the double equal also can be misleading if the integers are greater than 127 bits, because they may compare a number which is under 127 bits with something over, -128 bits, which complicates things and makes operations take longer.

Using key as a index

By adding a index to each element, or letting each zipcode be the index, we have successfully implemented a "hash table". Maybe not the most efficient one because we will need to increase the maximum array size significantly, from 10000 to 100000 to be exact. But lets see how much faster the lookup is. See Figure 1 for the lookup method used.

```
public static boolean hashLookup(Area[] array, Integer key) {
    if (array[key] != null && array[key].getPostnr().equals(key)) {
        return true;
    }
     return false;
 }
```

Figure 1: *Lookup method using key as index.*

After testing the lookup method in Figure 1, we get an instant response, 0 ns after testing. Compared to previous binary search this is not even comparable. We can expect the benchmark on the binary search, 10 times the array size compared to the previous, to be incredibly larger. Trying to test binary search for size 100000 in Visual Studio only caused crashes and was impossible, so it was skipped. Nonetheless, the same results and conclusions are made without the numbers.

But that error itself takes us to a very important subject, the drawback of using such large sizes. It is a huge waste when only using 10 percent of the size. So instead it is better to implement a hash function that uses modulo to map that index. But then we also get collisions, which will become apparent in next section.

Collisions

Collisions occur when two keys get mapped to the same index, or key. If we create a proper hash function by taking modulo each index and assigning that result to act as a index for a new array we will get some collisions, our goal is to minimize those collisions. Table 3 shows in what indexes collisions happen, depending on mod values used.

As we see in Table 3, it is logical to assume that a greater array will have less collisions. If we solely look at the largest number and the smallest, we see that there are 5 collisions for mod value "12345" while one less for "17389". But this is not always the case, because we are using a very simple

mod-val.	1st	2nd	3rd	4th	5th	6th	7th
12345	4997	1804	311	33	1	0	0
13513	5410	1678	287	12	0	0	0
13600	3406	1578	613	229	56	13	0
14000	3055	1316	615	320	134	31	1
17389	6352	1414	161	3	0	0	0

Table 3:
Indexes where we have collisions. After hash function.

hash function, the mapping heavily depends on the mod value itself, what numbers it consists of. Notice in Table 3 how mod value 13513, second smallest value, have collisions in 4 indices, same as the largest value.

Handling Collisions - Buckets and Linear Probing

When handling collisions, there are two approaches. One is to add ”buckets” where we ”toss” all elements with duplicate indices, and find them from there, and the other where we keep searching in the array for a better index for the said element. The second, which we can call Linear Probing, is more ”risky” in a way. Because if you have a very small array size, it will not quite work. Also if no indices are empty forward in general it will not work, unless you wrap. But the odds of there being absolutely no free elements are not great, so it should work nonetheless, especially in this case with zip numbers and an overhead of around 4000.
Lets look closer into how many times the probing needs to be done for different sizes. Here, the greater the array size, the less the program needs to probe as there are more gaps in between each element and much more overhead. See Table 4.

mod-val.	Elements Probed
12345	149640
13513	147356
13600	195099
14000	515942
17389	49442

Table 4:
Indexes and there probing numbers.

As we see in Table 4, it is not as simple as looking only at the size. Here again, because of the hash functions modolus dependency, it is all about what mod value is used. As we can see, a value like 14000, which we saw had the most collisions by quite a margin, it also needs to probe to a greater amount of elements. While a value like 17389, which was tied for the least collisions, as can be seen in Table 3 above, had way way less probing. This is due to two reasons. One, the size is much greater, leading to many more open slots and potentially greater gaps between each element, and less "traffic" in some areas.

The rest of the values are relatively similar, this is probably due to them being relatively close to their efficiency in distribution of values, while the value "12345" maybe being worse. If we ignore the "modulo efficiency", meaning we ignore the values that have great distribution or large gaps between collisions, we can see a clear pattern in Table 3 and 4, that values with more collisions and smaller size take the most time, while opposite is also true.