Assignment 9: Hash

Mahad Ahmed

Fall 2022

Introduction

In this assignment we will show the most natrual way of organizing a data set that should be accessible by a given key. In the beginning we implemented a not so effcient solution but then we tried to implement a better solution. This solution would give us a better space and time complexity. More on this later.

A table of zip codes

We were given a list containing Swedish zip codes using a CSV file. Each item i.e zip code would be an entry in a array. This was done by reading the file using the code that was provided.

Lookup

We wrote a method called lookup which is implemented as a standard search algorithm. By that I mean a linear search through the entire data entry until we find the specific entry we are looking for.

```
public boolean lookUp(String entry) {
   for (int i = 0; i<data.length; i++) {
      if(data[i] == null)
         break;
      if (data[i].code.equals(entry))
         return true;
   }
   return false;
}
```

Benchmarks

Part of the first task was to do a small benchmark comparing our linear search lookup method to binary search. First we searched for the zip code "111 15" which is which is the first entry in our list. Then we did the same for the last zip code which is "994 99". How did the two algorithms stack up?

Elements n	lookup	binary
111 15	199	799
994 99	156799	699

Table 1: Benchmark, lookup vs binary search. Zip as a string

The results are easily explained. When searching for the zip code "111 15" we see that the lookup method is very much faster than binary search. Because the "111 15" the first entry in our list which makes the lookup method find it in the first search. But binary search jumps into the middle of the list then working it's way down. But this is just one side of the coin. The other side when we are looking for the last entry in the list "994 99" then the lookup method must go through all the entries in the last. Binary search how ever can still just jump in the middle and then keep doing that until it can find the entry we are looking for. This is why the lookup method is linear as it goes through n elements until it finds what it is looking for. Binary search is a *O(nlogn)* procedure.

Now when we now that all the zip codes a number, then thy can't we just convert them into Integers before benching them. Before we had them as strings but now we have converted them into Integers. Let's see how the benchmark changes from the previous one.

Elements n	lookup	binary
111 15	99	199
994 99	49601	99

Table 2: Benchmark, lookup vs binary search. Zip code as an Integer

Now you can see that the algorithms are a bit faster than the previous benchmark. Why is that? Because now we don't have to convert them into integers which of course costs time.

How can we make it even better? Using keys as index can help with that. We know that the highest possible key is then 99 999 so we can just construct a array which holds a 100 000 elements and then we can use the indices as keys.

Elements n	**lookup**	**binary**
111 15	0	99
994 99	0	99

Table 3: Benchmark, lookup vs binary search. Zip as an Integer and using key as index

As you can see this lookup gives us a constant of 0. This could be also like 10, 200 and so on but as long as it is constant. As we already have the key which gives us the index of the entry we are searching for, we can find it directly. The time complexity is *O(1)*.

size matters

A drawback of the implementation we have done so far is that we now have a lot of space in our memory allocated but we only using about 10% allocated space. Meaning we have an array with 100 000 elements but we are only using about 10 000 of those. In small projects this can be ok, because we are only losing some bytes but in larger scale projects this would be a huge drawback.

Now for the solution which is to transform the key into and index in a smaller array. A function that does this is called a *hash function*. A simple of a hash function is taking the key modulo a value n and hope the indexes is somewhat unique [1]. When a key maps to the same index as another key, we call this a *collision*. This is something we need to handle later down the line. The fewer collisions the better.

Benchmarks

Let's now do some benchmarks where we test different values of n. We read the file and ran through them creating a index modulo n.

modulo n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
10000	4465	2414	1285	740	406	203	104	48	9	0
12345	7145	2149	345	34	1	50	0	0	0	0
20000	6404	2223	753	244	50	0	0	0	0	0
30000	7267	2010	397	0	0	0	0	0	0	0
50000	8491	1183	0	0	0	0	0	0	0	0

Table 4: Benchmark, where 1,2,3...10 is number of collisions of each type

Using a prime as a mod n then the collision become fewer. If we also increase mod n then we will also have less collisions, but doing this will make the array bigger which then also takes up a lot of space. Prime numbers are more efficient. Prime numbers are likely more efficient because they don't share common factors with any other value else except for 1.

handling collisions

So when dealing with collision i.e same string/integers that are mapped to the same index, how can we handle them? We can implement something called *chaining*. This method is basically when we have a collision we store them in a linked list. That means when we need a value in this index we may need to go through the entire or some of the linked list to find it.

```java
public void insert(Node nodeToInsert) {
    int index = nodeToInsert.code % size;
    Node current = data[index];
        if (current == null)
        data[index] = nodeToInsert;
    else if (current.next != null) {
        current = current.next;
        current.next = nodeToInsert;
    }
}
```

Benchmark

How does chaining help for the collision problem?

Elements n	**lookup**
11115	0
98499	0
12652	0
46197	0

Table 5: Benchmark, lookup with bucket/chaining

If we have a collision and we do a linked list then if we choose a large prime number we would only have maybe 2-3 at max 5 collisions. Meaning the most amount of nodes in a collision is five. Traversing through a linked list with five nodes is not that costly in the grand scheme of things. Which is why this is a constant lookup i.e *O(1)*.

Linear probing is another way of handling collisions. When you have a collision the instead of linking them as a linked list we instead search for the next available spot to the right of where we hashed. For example if we want to hash ”111 15” to index 13, we first look if the space is empty. In this case this space is occupied by ”123 45” so we look to the right of that zip code and if that is spot is empty we can put it there. The lookup for this is also constant time *O(1)*.

References

[1] Johan Montelius. Hash, 2022.