Hash tables

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the ninth assignment ”Hash tables” on the course ”ID2021” TIDAB, KTH. In this report the subject of hash tables, how they work, their strength but also when not to use a hash table. To explore the subject we will make use of a file with around ten thousand different postal codes to be the data that is being searched. A hash table or a hash map is a data structure like a dictionary that maps keys to values. A hash function is used to generate keys to the data values. Ideally each value gets a unique key, but this is most often not the case and when several values get the same key we need to handle this collision. To see the value of a hash map we will firstly search through the list of postal codes with a linear and a binary search function. We will then gradually implement a working hash data structure. The code for reading from the file and creating readable java objects from this is given in the assignment.

Linear and binary search

Linear and binary search is two search methods that was implemented in a previous assignment to use these in this assignment however they had to be slightly altered since they were implemented for a list of ”int” and now we will search through a file and create readable objects from it. The information from the file is collected and made into readable objects. To be able to use our previous search method implementations we use the built in operation ”equals” that compares if two objects have the same value rather than being in the same memory location. Another operation that is used is compareTo which is used to evaluate the Unicode values of ”String”, in this case since we have not yet converted all of the information to integers.

First we benchmark the linear and the binary search lookup methods for the first and the last entry in the file. With linear search the first entry took 115ns and finding the last element took 63000ns. With binary search finding the first element took 300ns and find the last one took the same time.

Now we change the way we save information from the file and convert it into "Integer" to see if there is any difference whilst searching. Now linear search for the first element took 120ns and for the last 30000ns. For binary search both the fist and the last now took 240$nstofind$. This is an improvement for when we compared number codes made that were "String" objects. The cause for this is probably that when comparing the magnitude of the string objects it takes more time to iterate through each element in the string object than when comparing "Integers".

Use a key as index

To try to improve on the execution time when searching through the postal codes in this file the task is now to make a very large array and then use the postal codes themselves as the key for where they are placed in the array. We then create a lookup method that simply takes in the code for the element and finds its value in the array. When bench marking this the lookup method took 80ns which is faster than the two other search methods, which was expected since we should get a constant run time complexity of O(1) since we simply go straight to an element in an array with its key and collect its value. The problem with this however is that we now have a very large array, in this case 100 000 elements with many array slots that are unused to make smaller and better keys since the postal code values were up to 10 000 we could generate a smaller value using some function.

Hash table

The function that is going to solve our problem is a hash function, it has to be quite simple since the point here is to make the search method fast and not slower. the function simply takes the code and a modulo value to decrease the value to a suitable size using modulo. See the code for this below:

```java
public static int hash(Integer code, int mod){
    return code % mod;
}
```

We want a hash function that creates as few collisions as possible but not a value so large that the the new array size will still be as large. To find a good value for this we do an experiment with a function that creates an array with our hash keys and then test how many collisions we get. To benchmark this we run this function with a few trial runs. The printout for this will show a row of numbers the first number show what modulo we are using then following in order the number of elements with 0 collisions, the number of elements with one collision, the elements with two collision and so fourth. The number has to be larger than the number of elements in our file since all elements need room. We can see from the benchmarks that an even and small number gives us many collisions and several multiple collisions. Prime numbers seem to give the best results which makes sense since we are using modulo. After bench marking 10 313 was the modulo value chosen to use in the rest of the code.

1. Printout with modulo 10 000

   10000, 4465, 2414, 1285, 740, 406, 203, 104, 48, 9, 0

2. Printout with modulo 20 000

   20000, 6404, 2223, 753, 244, 50, 0, 0, 0, 0, 0

3. Printout with modulo 12345

   12345, 7145, 2149, 345, 34, 1, 0, 0, 0, 0, 0

4. Printout with modulo 21313(prime number)

   21313, 8480, 1138, 55, 1, 0, 0, 0, 0, 0, 0

5. Printout with modulo 10313(prime number)

   10313, 6690, 2421, 488, 67, 8, 0, 0, 0, 0, 0

Handling collisions with buckets

The first way of handling collisions we make each element in the array into something we call a "bucket". Each bucket is a linked list with the size of one node, if we get a collision we extend the linked list with the new value that has been given the same key. To implement this we need a function that inserts a value to the array that takes this into consideration. In the case of out postal code file no value is the same but we will still cover this in the code. We first generate a hash key, we then check if there already is an element on the key position of the array we check if this value is the same as that we are trying to put in (will never happen with this data). To compare elements properly we also need the original code value before it is hashed to be able to make correct comparisons. If there is no value there we just add the data to the key position otherwise we iterate through the linked list until we reach the end and add the value there. The code for this is given below:

```
public void insert(Integer code, Node entry){
    Integer key = code % mod;
    Node curr = this.data[key];
    Node prv = null;
    while (curr != null){
        if (code.equals(curr.code)){
            curr = curr.next;
            break;
        }
        prv = curr;
        curr = curr.next;
    }
    if (prv != null){
        prv.next = entry;
    }
    else{
        data[key] = entry;
    }
    entry.next = curr;
}
```

We also need to implement a new lookup method that is adapted to this new data structure, the code for this is given below:

```
public boolean lookup(Integer key){
    Integer index = key % mod;
    Node curr = data[index];
    while (curr != null){
        if (key.equals(curr.code)){
            return true;
        }
        curr = curr.next;
    }
    return false;
}
```

Linear probing

Another implementation of collision handling is linear probing, instead of filling the array with "buckets" we could make use of the array itself. With this method, if we get a collision we simply iterate forward in the array until we find an empty slot for the value. This way when you want to access a key that has collisions you go to the array slot of that key and then iterate forward until you find your value. If you have an array with a modulo value that is close to the number of elements it could be problematic if the empty slots are too far away and it has to iterate through a large amount of elements. The implementation for this insert and updated lookup method is given below.

Inserting an element:

```
public void insert(Integer code, Node entry){
    Integer key = code % mod;
    while (this.data[key] != null){
        key++;
    }
    this.data[key] = entry;
}
```

Finding an element:

```
public String lookup(Integer code){
    Integer key = code % mod;
    while (!this.data[key].code.equals(code)){
        key++;
    }
    return this.data[key].name;
}
```

A benchmark method was made to benchmark this implementation. The maximal amount of steps that was made was 43. The average number of times we needed to take a step was 1.2 times and the average step was 7. When bench marking linear probing against the bucket implementation the bucket operation gave on average less comparisons which should give better execution time, the linear probing method however makes better use of memory in the machine. These two factors are good to keep in consideration when developing a hash data structure.