Hash tables

Axel Månson Lokrantz

Spring Fall 2022

Introduction

A hash table is a data structure that organizes data through a hash function. The hash function computes an index into an array of buckets from which a value can be found. When searching for an element the key is hashed into an index which indicates where the corresponding value is stored. In the following assignment we test different implementations and evaluate time and memory efficiency of a hash table. The program was written in Java together with Axel Lystam at KTH.

Task 1: A table of zip codes

A CVS file with all of Sweden's zip codes was given to have some data to work with. The first task was to create a linear and a binary search method for the data set.

```
    public void linearSearch(String entry){
        for (int i = 0; i < data.length; i++)
            if (entry.equals(data[i].code))
                return;
    }
```

Code section 1: Linear search for data set.

The method searches through the data until the specified entry is found. Best case time complexity is $O(1)$ and worst case is $O(n)$ when the element we are searching for is the last element of the array. A much faster search algorithm is binary search.

```
  public boolean binarySearch(String entry){
        int max = this.data.length - 1;
        int min = 0;

        while(true){
```

```
            int mid = min + (max - min) / 2;

            if(data[mid].code.equals(entry)){
                return true;
            }
            if(entry.compareTo(data[mid].code) > 0 && mid < max){
                min = mid + 1;
                continue;
            }
            if(entry.compareTo(data[mid].code) < 0 && mid > min){
                max = mid - 1;
                continue;
            }
            return false;
        }
    }
```

Since we are comparing Strings we have to use javas compareTo and equals method. The syntax "continue" acts as a recursive call. What caused the most problems in this implementation was the mid variable which calculates the center of the array. One could easily think that $mid = max+min/2$ would be sufficient, but half of the time we are dealing with an array of an odd length. The time complexity of binary search is $O(log(n))$. Best case occurs when the central index directly matches the searched element which results in $O(1)$.

Task 1.1: Benchmark

To benchmark our implementation we searched for two separate zip codes. An alternative implementation was also created where we rewrote the zip data as integers instead of strings.

Linear search	Binary search	Linear search (Int.)	Binary search (Int.)
42000 ns	1200 ns	33000 ns	900 ns

Table 1: Benchmark for lookup of zip codes through strings and integers. Measurements in nano seconds.

As seen in the table above, binary is a lot faster than linear search which was expected. We can also conclude that searching for a string is slower than an integer. This is because there are instructions on the machine level which can perform the comparison between integers in one cycle. For a string, which consists of several characters, you have to do several comparisons, which cost time.

Task 2: Use key as index

Since the zip codes are in order ranging from 111 15 in Stockholm to 984 99 in Pajala, why not create an array with 99 999 elements and use the zip code as their index.

```
public void linearSearch(Integer entry){
        if(data[entry] != null)
            System.out.println("Found! " + data[entry].name);
}
```

This implementation will result in a time complexity of $O(1)$, all we have to do is one comparison to see if the zip code is there or not. The problem is, however, that we have an array with more than 90% of the indexes which are empty. For smaller sets of data, this is fine, but for larger sets of data this could be a huge problem memory wise.

Task 3: Size matters

One way of solving the problem is to transform the key into an index in a smaller array. To transform the key we make use of a hash function. In this program we take the key modulo m to find a unique index. Depending on the number we chose as our m we might get collisions. A collision occurs when the hash function returns the same index for different keys. For example, if we were to chose 10 000 as our modulo number we would get around 4500 unique keys and nearly 2500 single collisions.

```
mod m: 10000,
unique: 4465
 1 collisions: 2414
 2 collisions: 1285
 3 collisions: 740
 4 collisions: 406
 5 collisions: 203
 6 collisions: 104
 7 collisions: 48
 8 collisions: 9
 9 collisions: 0
```

For a growing number of m we get fewer collisions but since the array of data has to be of the length of m it is a trade off between size and collisions. It turns out prime numbers work well when choosing a value for m. In this program we chose m: 24979 which gave us the following results.

```
mod m: 24979 (prime)
unique: 7974
```

```
1 col: 1556
2 col: 143
3 col: 1
4 col: 0
5 col: 0
6 col: 0
7 col: 0
8 col: 0
9 col: 0
```

Task 4.1: Handling collisions

There are several ways of handling collisions, one is to create an array that holds buckets where each bucket holds the elements with the same keys.

```
public Integer getHashCode(Integer zipCode, Integer mod){
    return zipCode % mod;
}

public void insertLinked(Node node, Integer mod){
        Integer index = getHashCode(node.code, mod);
        if(hashTable[index] == null)
                hashTable[index] = node;
        else{
                Node current = hashTable[index];
                while(current.next != null)
                        current = current.next;

                current.next = node;
                }
}
```

When creating the array we first retrieve a hash code which corresponds to the index of the element we want to add. If the index is empty, we simply insert the element. Else we traverse the list up until the last element at which we insert the node.

Task 4.2: Slightly better

The problem with the implementation at 4.1 is that there will be a lot of empty indexes. A slightly better implementation is to make an insertion method that looks for a free slot in the array starting from the hash index. The lookup procedure ends as soon as the method finds an empty index.

```
public void insert(Node node, Integer mod){
        Integer index = getHashCode(node.code, mod);
        if(hashTable[index] == null)
            hashTable[index] = node;
                else{
                  for(int i = 1; i < hashTable.length; i++){
                         if(hashTable[index + i] == null){
                                hashTable[index + i] = node;
                                return;
                        }
                }
        System.out.println("Not able to add element.");
        }
}
```

With mod 24 877, we ended up with 16452 empty indexes with the insertion method in 4.1 and 15202 empty indexes with insertion method above.

Task 5: Benchmark

To evaluate the implementations we created a two lookup methods that calculated the average comparisons needed to add an element with a growing mod value.

mod	comparisons
15000	1.703
20000	1.481
25000	1.363
30000	1.289
40000	1.209
45000	1.182
50000	1.141

Table 2: Benchmark for insertion in a hash table with linked lists.

mod	comparisons
15000	53.931
20000	17.632
25000	11.126
30000	6.508
40000	4.986
45000	3.015
50000	3.009

Table 3: Benchmark for insertion in a hash table without linked lists.

All in all, we can conclude that the bucket implementation that utilizes a linked list structure is less memory efficient but more time efficient since the number of comparisons are far fewer than the second implementation.