Hash tables

Jonathan Nilsson Cullgert

Fall 2023

Introduction

In this assignment we will explores hash tables. To start of some inefficient ways are explored and lastly a good solution will be implemented. A hash table is a way of storing data by changing the key of the inputted with some function and then using that hashed value as the index in an array. As an example if the key of the data is 111 i change it in a hash function so it becomes 1, the 1 is then the index in my hash table. Since multiple keys can get the same hash key, if i put 222 through my hash function i might also get 1 as the new key. These two will cause a collision and being able to handle such an event is important, which will be touched upon later.

A table of zip codes

In this assignment a csv file will be read and stored in nodes, this is the data that will be handled and eventually put into the hash table. The important part here is to make sure that the csv file is put into the correct file path where the java program is being executed from. If the file path is not correct the file will not be read in to the program and will subsequently not be able to be handled. The nodes in which the data will be stored in will contain a string of the zipcode, a string of the name and an integer of the population.

An array is created where every element in the data set is put at the index which corresponds to their place in the csv file. So the first line of data is put at index 0 in the array and so on.

A **lookup()** method is created which will linearly look through the array and by using the **equals()** method in java will compare a given zip code to all the zip codes in the array. As one might expect this has a time complexity of $O(n)$. Since all the zip codes are sorted from the beginning a binary search can be implemented. Since strings are being compared the **compareTo()** method can be used to know which side of the middle element that has to be ignored. This has a time complexity of $O(log(n))$. These results can be seen in table 1. They might not behave as expected but I believe it is because it is benchmarked on small n.

The next part of the assignment was to exploit the fact that zip codes are integers and by changing the node to contain an integer of zip codes, string of the name and an integer of the population. This change is made by this code snippet.

```
Integer code = Integer.valueOf(row[0].replaceAll("\\s",""));
data[i++] = new Node(code, row[1], Integer.valueOf(row[2]));
```

Now the zip codes can be compared with standard operators $< \mid = \mid >$ and therefore the binary and linear search are benchmarked again but with these changes in mind. As we can see in table 2 the linear search becomes extremely slow but binary search becomes even faster. I am not certain as to why the linear search becomes so much slower since the **equals()** method should be a slower way to compare the zip codes than by just using standard mathematical operators. The first lookup method might not have searched through the entire array or something else is simply coded wrong, such things happen. The binary search behaves as expected, keeps the same time complexity and becomes faster since it does not have to use the **compareTo()** method which is slower than comparisons.

Use key as index

The next part of the assignment is to just the zip codes as keys and indices. Since the zip codes are perfectly good numbers and no two zip codes are the same might as well use them as indices to the array. When the lookup method is used on this implementation we see that the linear approach is faster. That is because its time complexity is now $O(1)$, we just have to check the index of the zip code that we are given, if it null not there otherwise the zip code exists.

The problem with this implementation is that the we have about 10 000 elements in the csv file, if the zip codes are meant to be used as the keys we will have to a lot more memory since the zip code go up to 100 000. Not only does the array have to incredibly large almost none of it is being used, only a measly 10% is being used by the data. So while it is incredibly fast it is using a lot of memory unnecessarily which might not be a problem in this assignment but in general real life applications it certainly will be.

Size matters

To fix this problem the hash will be implemented, at least the beginning of one. If we take the zip code, key, from our data and put them in to a function to change their value in to a new key in a smaller range. This is called a hash function and such a function should not be overly complicated since it is meant to be fast. A simple way which will be used in this assignment is

to take the key modulo some other value m and then it will be a new key, a hash key. Using this method it is obvious that some key will get the same hash key and if that happens a collisions will take place. In this part of the assignment the collisions are not handled just saved.

```
public void collisions(int mod){
    int[] data = new int[mod];
    int[] cols = new int [10];

    for (int i = 0; i < max; i++){
        Integer index = keys[i] % mod;
        cols[data[index]]++;
        data[index]++;
    }

    for(int i = 0; i< 10; i++){
        System.out.println(cols[i] + "\t");
    }
    System.out.println();
}
```

This piece of the code will take in an integer *mod* which will be the modulo operator. It will then create a new array in which it will increment the index the hash key is pointing to. If the index has 5 hash keys pointing to it, it will then increment cols at index 5 by one. By running this through with all our keys it will show with the array cols how many times hash keys will be colliding. As an example with $mod = 12345$, there will be 7145 hash keys not colliding, 2149 colliding once, 345 colliding twice and so on. An average collision can calculated with this code.

```
while(cols[maxCollisions] > 0)
    totalCollisions += cols[maxCollisions] *maxCollisions++;
float averageCollisions = totalCollisions/(float)max;
```

On average $mod = 12345$ will have 0.3 collisions.

Handling collisions

So now that we know that collisions will happen how do we handle them. First of all it is important that we want as few collisions as possible, therefore a good modulo should be used, for example $mod = 12345$. Now if we get collision we do not want to replace some data that is at a certain element but we still want two hash keys to be able to point to the same index. If we create an array at every index of hash we can then just place the data from the hash key in the first available spot in the array. This is called a

bucket. Since we do not want to use unnecessary memory implementing it as an array will be costly since we will have to copy elements and so on if we want to make it bigger, so why not use a linked structure.

In this implementation we will have an array of buckets. The buckets themselves will contain a node as well as a next pointer to another bucket. This way if a key turns in to a a hash key, lets say 15. We then store the data from the first key at index 15 as a bucket. We then have a new key with the hash key 15, we then store the data from that key by letting the bucket point to new data, which is also stored as a bucket.

```
public class Buckets{
    private Node code;
    private Buckets next;

    public Buckets(Integer code, String name, Integer pop){
        this.code = new Node(code, name, pop);
        next = null;
    }
}
```

This way we will be able to handle collisions if they do happen. The good thing with the hash table is since the hash keys will be generated in such a way that there will not be many collisions, finding, removing or adding elements to the hash table can be done in $O(1)$ time. The trade off with having this implementation is that the array where the buckets are stored will be bigger if the modulo operator is bigger. It will be a trade off between memory and time. For example if the modulo operator is low like 10, the time complexity to search for some data will become $O(n)$, since it will be just like linearly searching through a linked list.

Slightly better?

Another way to implement the hash table is to use an array. We start at the hashed key and then move forward in the array until an empty spot has been found. This is quite a simple way of implementing the hash table. Since will probably not have to look through too many elements before we find the one we are looking for.

It might be obvious that this way of implementing the hash will not be that effective if the array we are storing in is small or if the amount of elements being stored is larger than the array. If the array becomes close to being full it will perhaps have to iterate through a lot of elements before finding the one it is searching for. If we have an array of 100 elements that is full except for at index 99. If we then pass through a key with the hash key 0, we will then have to iterate through the entire array and then place

that data at index 99 even though the hash key is 0. This is clearly quite ineffective which means that this implementation is only useful with large arrays and when they are not close to being full.

With the previous example of using $mod = 12345$ the hash key only had to move down 38 places in the array on average. If the size of the array is increased that number did not change which quite obvious since the hash function will still produce the same hash key. If a lower modulo operator was used in a large array, the average amount of steps data would have to travel would decrease if the size of the array was increased.

Benchmark

n	linear	binary
100	700	200
200	400	100
400	400	200
800	300	300
1600	200	500
3200	300	500

Table 1: Table to find "111 15" and "984 99" zip codes as strings n times in us

n	linear	binary
100	7000	70
200	3000	110
400	5000	40
800	12000	60
1600	20000	90
3200	43000	180

Table 2: Table to find 11115 and 98499 zip as Integers n times in us

n	linear	binary
100	90	20
200	20	50
400	20	80
800	30	30
1600	60	100
3200	120	160

Table 3: Table to find 11115 and 98499 as integers with keys as index in us