Hashtables

Aditya Gupta

October 2022

1 Introduction

So far we have looked at many different data structures such as arrays, linked lists, trees etc. All of these data structures have varying degrees of efficiency when one tries to search for a key value in this data set. Here we look at a data structure which arranges data in the most natural way so it's easy to access any of the entries using a key.

2 Zip Codes

We first implement a hashtable which will be used to store information associated with certain zip codes. We implement this using an array of Nodes, where a node just refers to an object that contains all the attributes we want to store at a particular index. Every node contains the Zip Code and the area associated with the code as a String while the population at each of these locations is stored as an Integer. The internal array of the Hashtable has been set to a maximum capacity of 10000 entries. After this, we enter all the values from the CSV file provided to us for the assignment into the data array using a Buffered reader.

2.1 Linear Lookup:

Here we do a linear search for the zipcode data inside the data array. We do this by comparing the string the user is looking for and comparing it against all the entries in the data array and once we find a node with the right zip code, then we can return it to the user. If the requested zip code does not exist in the database then we just return null.

```
public Node lookupLinear(String s){
    for (int i = 0; i < data.length; i++) {
        if(s.equals(data[i])){
            return data[i];
        }
    }
```

```
    return null;
}
```

We can then change the code such that the zip code attribute is an Integer type rather than a String. We then do the same test but s is now an integer instead of a string. After this, we run some benchmarks to compare these 2 cases.

The benchmarks for both cases are given below:

String Linear Lookup	Integer Linear Lookup	Ratio taking integer lookup as reference
10740	350	31

String Binary Lookup	Integer Binary Lookup	Ratio taking integer lookup as reference
203	47	4

We observe that in the case where we represent the zip code as a string it takes a lot longer compared to the time taken in the case of an integer zip code. This is because comparing strings is more expensive as we compare each character in both strings. Thus the integer zipcode case performs a lot better.

2.2 Using key as index:

Since we know that the largest zipcode value is within the range of 100000, we replace our data array with a data array with the size of 100000. This means that We can now access the nodes directly as we can now use their zip codes as an index for the data array.

```
public Node lookupConstant(Integer s){
   return data[s];
}
```

The time taken to search for a specific data point using this approach shows a constant time trend as we have direct access to each node now. This implementation is very fast but extremely space inefficient as we never use 90 % of the array for anything. This is not very wasteful right now as we are only using 100000 spaces, but for a larger data set a lot more space might have been wasted that could have been better utilized by some other task on one's laptop.

Benchmark comparing Binary Search to using the zip codes as index:

Zip code as index	Binary Search	Ratio taking direct access using zip code as reference
38	547	14

3 Collisions and handling them:

Now we want to use a smaller array as we end up wasting a lot of space given that we only have 9675 values, and we are using a 100000-sized array. We will make use of something called a hash function to map the zipcode values to indexes in an array of a smaller size. In this assignment, our hash function is very simple which is given by the formula: `index = (zip code)%(array size)` (Here the % symbol indicates the modulus operation). As our hash function is quite simple we can expect to observe some "collisions". A collision means when more than one zip code is mapped to the same index value. In this part of the report, we analyse the number of collisions with the size of the array(as the hash function performs a bit better depending on the value of the size of the array). Using the code we were given inside the assignment document we get The number of collisions by key i.e. the number of instances where some keys map to the same index. For different-sized arrays, we have a different amount of collisions.

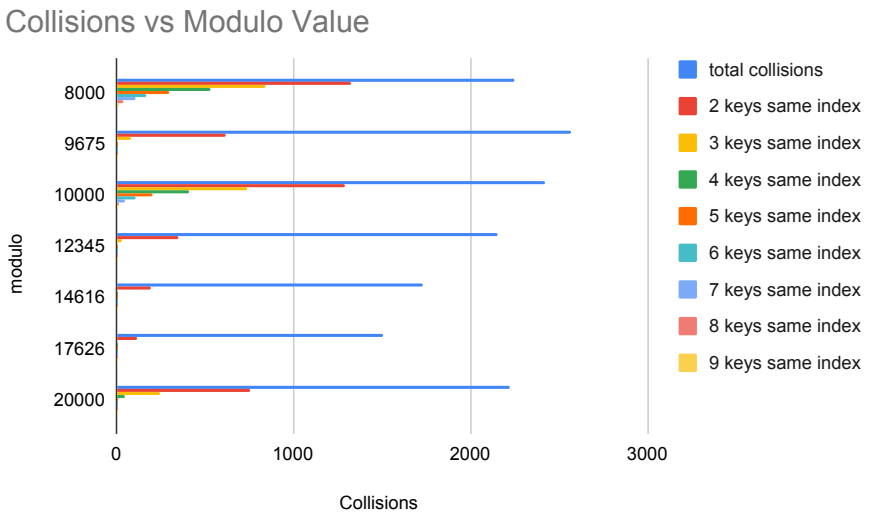

Figure 1: Graph for the Collision depending on the choice of modulus

In the above graph, we see the number of collisions observed and of what type. By that, I mean how many keys map to the same index. So for example for a modulo value of 10000, there are 1285 indexes where we see that 2 keys map to the same index. We observe that the number of collisions decreases as the modulo value increases. But we also notice that the largest modulo value doesn't necessarily lead to the least number of collisions as we see that for some numbers the amount of collision is a lot less than the bigger numbers.

3.1 Dealing with Collisions:

There are 2 possible methods for dealing with collision:

- Chaining
- Open Addressing

To deal with collisions using chaining, we add a next pointer attribute to the node class, which is used for storing a colliding object at the next position of the object that already existed at that position. This creates a linked structure at each position with a collision and then we traverse this linked list to find the exact object we were looking for. Depending on the competence of our hash function the linked list created at each position should not be too big.

While in the case of open addressing nodes that were assigned the same index by the hash function are stored at the next available index inside of the hashtable. When we want to search for this index we do a linear search for the node from that index onwards.

3.2 Chaining:

To be able to use the nodes at each index as a linked structure I add a next pointer attribute to the node class. If we encounter a collision we traverse through the linked list and add the new node to the end of it. When we search for an element using a zip code, we first use the index() function to retrieve the index for this specific zip code.

```
public String lookupWithChaining(int zip){
    var index = this.index(zip);
    if (data[index] == null)
    return "No such address";
    var node = data[index];
    while (node != null && node.zipCode != zip) node = node.next;
    if (node != null && node.zipCode == zip) return node.toString();
    return "No such address";
}
```

The time taken to do the above search operation takes close to 25 times more time compared to doing the search operation inside of the 100000-size array.

100000 Lookup	Lookup with Chaining(modulo = 17626)
38	969

3.3 Open Addressing:

Whenever a collision event is observed we move forward in the array until we find an empty space to put our node inside of. To search for an element we

calculate the index the same way as before using the index() function. This also means that when we search for an element when using this approach as soon we reach an empty position while searching inside of the array that means the node that one is searching for in this array does not exist. If we have traversed the entire array without a match then the object certainly does not exist in this hashtable.

```
public String lookupOpenAddressing(int zip)
{
    var index = this.index(zip);
    if (data[index] == null) {
        return "No such address";
    }
    var i = 0;
    while (data[index] != null && zip != data[index].zipCode && i < modulo)
    {
        index++;
        index %= modulo;
        i++;
    }
    if (data[index] != null && zip == data[index].zipCode) return data[index].toString();

    return "No such address";
}
```

The time taken to do above search operation takes close to 21 times more time compared to doing the search operation inside of the 100000-size array.

Modulo	100000 Lookup	Lookup with Open Addressing
10000	38	1136
14616	38	893
17626	38	799
20000	38	840