Hash tables

Dhruba Das

Spring Fall 2023

Introduction

This assignment introduces us to the idea of using hash tables, and using the hash function to allocate spaces for elements in an array, using a hash index produced by said function. There are mainly 4 problems in this assignment: comparing the speed of binary and linear functions which have either Integer or String arguments, using hash index and benchmarking lookup, implementing and benchmarking a 2D hash table, and implementing and benchmarking a 1D hash table.

Zip vs Zap

Here, I will only be going through the methods implemented in Zip, and then list the changes which were made in the Zap file.

Linear and Binary

This method linearly searches through the array, looking for the node which has the same zip code as the one we're looking for. Below is the code snippet:

```
public String linear(String zip){
        for(int i = 0; i < data.length; i++){
            if(zip.equals((data[i]).code))
              return (data[i]).name;
        }
        return null;
    }
```

The Binary method simply does the same as the linear search method, except using the binary search algorithm.Below is the code snippet:

```
public String binary(String zip){
        int mn = 0;
        int mx = max;
```

```
        while(true){
            int index = (mn + mx)/2;
            int cmp = zip.compareTo(data[index].code);

            if(cmp == 0){
                return data[index].name;
            }

            if(cmp > 0 && index < mx){
                mn = index + 1;
                continue;
            }

            if(cmp < 0 && index > mn){
                mx = index - 1;
                continue;
            }
            break;
        }
        return null;
    }
```

The only difference between the Zip and the Zap file are the arguments for the *linear* and *binary* methods; in the Zap file, the arguments for the *linear* and *binary* methods use Integer zip, instead of String zip.

Benchmarks

Below are the benchmarks for Zip (String zipcode):

Zip code	**Linear(ns)**	**Binary(ns)**
111 15	125	208
984 99	13292	208

Table 1: (For each code, a thousand tries are done; min. time taken; time in nanoseconds)

And below are the benchmarks for Zap (Integer zipcode).
As we can see, using an Integer zip is faster than using a String zip:

Zip code	Linear(ns)	Binary(ns)
111 15	83	125
984 99	7541	125

Table 2: (For each code, a thousand tries are done; min. time taken; time in nanoseconds)

Using key as index

This was quite easy to implement; only changes that were made were the initialisation of the array with 100000 elements, implementing the lookup function, and using key as index to fill up the array. Below are the changes:

```
public Zop(String file) {
        data = new Node[100000];
        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
            String line;
            int i = 0;
            while ((line = br.readLine()) != null) {
                String[] row = line.split(",");
                Integer code = Integer.valueOf(row[0].replaceAll("\\s",""));
                data[code] = new Node(code, row[1], Integer.valueOf(row[2]));
            }
            max = i-1;
        }
        catch (Exception e) {
            System.out.println(" file " + file + " not found");
        }
    }

    public String lookup(Integer indx){
        if(data[indx] == null)
          return null;
        return data[indx].name;
    }
```

Here as we can see, the lookup function uses the index of the zipcode to search for the node with that zipcode.

Benchmarks

Below are the times for the lookup function for the two provided zip codes:

As we can see, the lookup function is much faster than binary search.

Zip code	Lookup (ns)
111 15	83
984 99	83

Table 3: (For each code, a thousand tries are done; min. time taken; time in nanoseconds)

Hash tables

When implementing the hash table, we make use of 2D arrays, and implement a hash function. In this implementation, we have an array of buckets; in each bucket, there is only one zip code; each zip code has it's own unique hash index; this is done to avoid collisions (i.e when two zip codes have the same hash index). Below I will be going through the new methods which have been implemented in this file.

hashInteger method

This method gives us a hash index for a given zip code. Below is the code snippet:

```
public static Integer hashInteger(Integer zip, Integer range){
        return zip % range;
    }
```

addToBucket method

This method adds a node to a bucket, using it's hash index; if the bucket is full, it's extended by one spot:

```
public void addToBucket(Node n){
        Integer index = hashInteger(n.code, mod);
        if(hash_table[index][0] == null){
            hash_table[index][0] = n;
            return;
        }
        if(hash_table[index][0] != null){
            Node[] newBucket = new Node[hash_table[index].length + 1];
            for(int i = 0; i < hash_table[index].length; i++){
                newBucket[i] = hash_table[index][i];
            }
            newBucket[hash_table[index].length] = n;
            hash_table[index] = newBucket;
        }
    }
```

lookup method

This method is the same as the previous lookup method, but instead looks through a bucket using the node's hash index:

```
public String lookup(Integer zip){
        Node [] bucket = hash_table[hashInteger(zip, mod)];
        for(int i = 0; i < bucket.length; i++){
              if(bucket[i] != null && zip.equals(bucket[i].code))
                return bucket[i].name;
        }
        return null;
    }
```

Best Modulo

I used the given collisions method to determine the best modulo for the hashInteger function; I found it to be: 13513 7387 1976 299 12 0 0 0 0 0 0 The first number represents the modulo, and the subsequent numbers represent the number of collisions in each bucket, with 7387 collisions in the first bucket, 1976 collisions in the second, and so on.

Benchmarks

Below represented are the times for the lookup function, implemented using a 2D hash table:

Zip code	**Lookup (ns)**
111 15	42
984 99	42

Table 4: (For each code, a thousand tries are done; min. time taken; time in nanoseconds)

Better

In this implementation of the hash table, everything is the same as the previous one, except we use a 1D hash table and add nodes to the next free index, instead of increasing the size of the bucket.

Insert method

This method adds a node to the array using it's hash index, but instead of increasing the size of the bucket, we increase the hash index to the next one

in the array, in case of a collision:

```
public int insert(Integer code, Node entry){
        Integer indx = code % mod;
        int collisions = 0;
        boolean p = false;
        while(true){
            if (data[indx] == null){
                data[indx] = entry;
                break;
            }
            indx = (indx + 1) % mod;
            collisions++;
            p = true;
        }
        if(p)
          System.out.println();
        return collisions;
    }
```

Lookup method

This method is the same as the previous lookup method, but it now returns the number of elements we have to go through before finding the the node with the correct zip code:

```
public int lookup(Integer zip) {
        Integer indx = zip % mod;
        int elementsChecked = 0;
        while (true) {
            if (data[indx] == null)
                break;
            if (data[indx].code.equals(zip))
                break;
            indx++;
            elementsChecked++;
            if (indx == mod)
                indx = 0;
        }
        return elementsChecked;
    }
```

Benchmarks

Here, we look at the no. of elements we must go through before finding the desired zip code, for a number of zip codes:

Zip code	**No. of elements gone through**
111 15	0
352 46	0
591 62	3
737 91	10
984 99	16

Table 5: (For each code, a thousand tries are done)

Although not relevant, I also found this implementation of the hash table to have a slower lookup time in general when compared to the 2D hash table implementation.