Hash tables

Fatima Mohammad Ali

Fall 21/10/2023

Introduction

Hashing is a technique that maps a given key to another value. If two keys were mapped to the same value, a *collision* would occur, which could impose data-managing problems. There are several ways to solve the collision issue, and the one chosen in this assignment was *bucket hashing*. Hence, different reading procedures and collisions were the main focus of this assignment.

A table of zip codes

To start this work as a whole, a file of Swedish zip codes was given to be read in the program. Each entry contained a *zip code* (sorted numbers starting at 111 15), the *name* of its area, and the *population*. To read those accordingly, a `Node` structure was initialized inside a `Zip` class. The `Zip class` kept track of a `Node[] data` array and an `int max`.

Implementation one - Zip

For the first `Zip` implementation, the `Node` class had the properties: `String code, String name` and `Integer pop`. After reading the file and adding all entries to a `data` array (size = 10 000), a simple `linear(String zip)` method respectively a `binary(String zip)` method was implemented. Those had traditional algotihms that compared two *Strings* with the built-in `equals` and `compareTo` methods.

The benchmark results of the linear and binary search methods (of the first and last zip codes "111 15" and "984 99" as Strings) are given in table 1.

Implementation two - Zap

Changing the zip code from type `String code` to `Integer code` read the zip codes as *integers* to the `data` array. With the same `binary` and `linear` search methods as `Zip`, the following benchmark results were given:

Zip lin "111 15"	Zip lin "984 99"	Zip bin "111 15"	Zip bin "984 99"
170	81000	2300	960
Zap lin 11115	**Zap lin 98499**	**Zap bin 11115**	**Zap bin 98499**
280	49000	1800	380

Table 1: *T*imestamps of 1000 respective searches given in microseconds and rounded to two sign. figures.

The values of `Zip` showed several things. The linear search stopped the search when finding the wanted value. Therefore, the search was done after one comparison of Strings which returned the smallest time. Looking for the last element element however, required comparing all Strings to each other, which makes the time to retrieve the last String the longest. Binary search on the other hand, showed good time for searching for the last value. Finding the first zip code still required *O(log(n))* time instead of constant time as done in the linear search, which is why it is more time-consuming than its counterpart.

The results of `Zap` showed significantly improved timestamps compared to `Zip` values. This could be due the fact that comparing two *integers* is less time-consuming than going through and comparing each character of two *Strings*.

Implementation three - Zop

Since the `Integer code` is now an integer, it would be interesting to use the code itself as an index of an array. This was implemented in a `Zop` class and required the `data` array to be of size 100 000, since the largest possible key would be 99 999. A simple `lookup(Integer key` was implemented. This method solely checked if the `data[key]` was null (then return null), otherwise return the area name. Getting the timestamps for the first and last key searches gave the following timestamps:

Zop lookup index 11115	Zop lookup index 98499
280	83

Table 2: Timestamps of 1000 respective searches given in microseconds and rounded to two sign. figures.

The results of `Zop` are significantly faster than `Zip` and `Zop` since each index in an array is accessed in constant time. There is no need to iterate through all zip codes to find the desired one, the name is already given in the index. The downside of this implementation though is that the big array is mostly empty (all non-zip code indices are null).

Hashing and collisions

To solve the problem of a big array being empty, compressing all values to a smaller array could be the solution. To transform the original key into an index of a smaller array (smaller array could be ranged 0-10 000), a *hash function* was needed. A simple hash function transforms a key by calculating it modulo m, which gives a new value that is *hopefully* unique. If two keys map to the same index however, a so-called *collision* would occur.

The assignment provided a `collision(int mod)` tester method that computed the following table given a module value m:

mod is 10000:	4500	2400	1300	740	410	200	100	48	9	0
mod is 20000:	6400	2200	750	240	50	0	0	0	0	0
mod is 12345:	7100	2200	350	34	1	0	0	0	0	0
mod is 13513:	7400	2000	300	12	0	0	0	0	0	0
mod is 13600:	5900	2500	910	300	69	13	0	0	0	0
mod is 14000:	5500	2400	1100	490	170	32	1	0	0	0

Table 3: Each column is rounded to two sign. figures and shows the number of collisions of each key type, e.g. two keys mapped to the same index, three keys mapped to the same index etc.

When hashing values modulo m, there is something specific that should be considered to obtain the best possible hash values. Each new hashed index should be fairly unique. If m is chosen to be a primary number, then the keys would be divisible by one factor which would avoid multiples of other factors. In row 4 of table 3, it can be seen that most of the keys of m = 13513 are mapped alone in an index. No four keys with the same hashed values were mapped to the same index etc., which shows that the hashed indices are fairly distributed along the hash table. On the contrary, hashing values modulo 10 000 resulted in multiple collisions of the same type throughout the table.

Furthermore, it can be noted that having a bigger module m that creates a bigger hash table does not guarantee fewer collisions. As seen in rows 13600 and 14000, these m values are larger than 13513, but more collisions are still occurring. This is since these modulo values are not relatively prime and would compute similar hash values for different keys.

Handling collisions - array of buckets

In this assignment, *bucket hashing* was chosen to solve the collisions issue. As a start, it is worth mentioning that the reading file procedure was done in similar manner as the `Zop` class. The zipcodes were indices of a `data` array of size 100 000 and each node held an `Integer code`, `String name`, and `Integer population`.

The implementation of a bucket was done by initializing a `Node[] bucket` in the `Node` class. Whenever a bucket was needed, a `bucket` of primary size 1 was created. Hashing keys was done in a `hashKeys(int mod)` method outside the `Node` class. This method declared a hash table as a `Node[] indxs = new Node[mod]` array (at last, it was set as the `data` array). If the hashed node of the `indxs` array was null, then the indxs pointer would point to the `data` value (`indxs[indx] = data[keys[i]];`. If the node was not null, meaning it already had a hashed value on it, the following `insertBucket(Node key)` was called:

```
//in the Node class, initializes and adds to bucket
public void insertBucket(Node key){
    if(bucket == null)  //if the bucket is not initalized
        bucket = new Node[1]; //create bucket
    //if the bucket is full, extend
    if(bucket[bucket.length - 1] != null){
        Node[] newbucket = new Node[bucket.length*2];
        for(int j = 0; j < bucket.length; j++)
            newbucket[j] = bucket[j];
        bucket = newbucket;
    }
    //insert key in bucket
    for(int i = 0; i < bucket.length; i++){
        if(bucket[i] == null){
            bucket[i] = key;
            return;
        }
    }
}
```

To check whether the bucket worked as it should, a global counter was used to return the depth of a bucket when hashing the same key a number of times. For instance, hashing the key 98499 a number of 25 times would call the `insertBucket` method 24 times, which showed the depth of a bucket.

Furthermore, it was needed to check whether the buckets contained the correct information of the zip code. To do so, a lookup method was implemented. A `String searchBucket(Integer zipcode)` method was implemented in the `Node` class. This method solely iterated through a bucket until the given *zip code* was found and its *name* was returned. Outside the `Node` class, the `searchBucket` method was called in a `String lookup(Integer zipcode, int mod)` method. The method hashed a value for the given zipcode and checked the `data` array. If `data[indx]` was null, then there was no value to return. If the zipcode was equal to a code of the hashed index of `data`, the name was returned. Else, `data[indx].searchBucket(zipcode)` was called and a bucket was searched through.

Checking whether the lookup method retrieved names from the hash table was done by first hashing the keys (calling `hash.hashKeys(mod)` in main) and then looking up a certain key. For instance, calling `hash.lookup(52025, mod)` would give "DALUM" as returned value, which is correct. When trying for different modulo sizes, the same results was given. Therefore, the lookup method worked as it should.