Hash Report.

Adrian Jonsson Sjödin

Fall 2022

1 Task

In this task we will read a file in CSV format that contains zip codes in an ordered format, and then look for specific entries using different methods.

- Start with writing a linear lookup method and a binary lookup and write a small benchmark that searches for "111 15" and "994 99" and explain the results. Then change the node class so that it holds an integer as code and re-run the benchmark. Explain why the execution time has improved
- Use the zip codes as keys and store them in an array, using the keys as indexes in the array. Implement a lookup method and compare its execution time to the binary search method in the previous class.
- Take all keys and convert them to a hash and store them in a array. This can be done by taking the key modulo m for some value m. Count the number of collisions for each type and see which m seems to give the least number of collisions. To be able to handle the collisions first implement "buckets" where the buckets are a separate array. Then implement an improved version where the bucket uses the array with hashes itself. Gather some statistics on how many elements you need to look at before you find the one you're looking for. Compare the result with the separate array bucket implementation.

2 Method & Theory

The code for how to read the CSV file and store it in an array was provided in the assignment description, so all that was needed to implement for the first task was the linear lookup and the binary lookup. The linear lookup is just iterating through the data array and for each entry check if the code field in the entry is the same as the one we're searching for. This is done using `.equals()` and when the correct entry is found we simply return the name field. This method will give us a time complexity of $\mathcal{O}(n)$.

The binary lookup method works as in the previous assignment. We start with going to the middle of the array, check if that is the entry we're looking for and return it if it is. Else we check if the entry we're looking for is higher or lower than where we're at and adjust our min, max borders accordingly before going to the middle of the new interval and repeating the process again until we find the entry we want. The checking of entries are done using the `.compareTo()` method that compares two strings lexicographically. This comparison is based on the unicode value of each character in the strings. The time complexity of this method should be $\mathcal{O}(log(n))$

Changing so that the code field is an integer only required a slight modification, already given in the assignment instructions. This in turn only required that I changed the parameter from `String` to `Integer` in both the linear and binary lookup method, since both `.equals()` and `.compareTo()` also works for integers. The time complexities should remain the same.

Using the zip-codes as keys simply required me to allocate a large enough data array and then change the line `data[i++] = new Node(...);` in the constructor to `data[zipCode] = new Node(...);`. Writing the lookup method is now simply returning the entry at the index of the zip-code we provide it. The time complexity of this method is $\mathcal{O}(1)$

Lastly we where supposed to convert the zip-code to a hash and use this as the index position in the array where we save the node. The hash function itself was really simple and consisted of simply taking the zip-code modulo a prime number. This will allow us to use a smaller array than in the method above, however we will have collisions since some zip-codes will produce the same hash. In the first implementation this is solved by letting every entry node be able to point to another node, creating a small linked list on those indexes where we have a collision. So when we input the nodes in the data array we first check if it is not empty at the hash index we want to place the node. If that's the case we check that entry's next pointer and loop through the linked list until we reach the end where we then place the new entry. And in the case there's nothing at the hash index already we can simply place it there directly.

The lookup method then becomes real simple. Just take the key parameter and convert it the the hash index using modulo and check the entry at that index position. If there's a small linked list there then simply iterate through it until we find the correct node. This can be seen in the code bellow.

```java
public String lookup(Integer key) {
    Integer index = key % this.mod;
    Node current = data[index];
    while (current != null) {
        if (key.equals(current.code))
            return current.name;
        current = current.next;
    }
    return null;
}
```

The second implementation made used of the empty spaces that where in the data array. Instead of using a linked list on those indexes where we had collisions, we instead just incremented the hash index with one until we reached a spot in the array that where empty. The lookup method in this case became even simpler since all we needed to do was get the hash index from the code parameter and then step through the array starting at the hash index and increment it with one until we got the correct entry. This can be seen in the code bellow.

```
public String lookup(Integer code) {
    Integer key = code % this.mod;
    while (!this.data[key].code.equals(code)) {
        key++;
    }
    return this.data[key].name;
}
```

3 Result

Table 1: Linear search and binary search for when the zip code is in `String` format

linear 111 15	Stockholm	92 ns
linear 984 99	Pajala	32224 ns
binary 111 15	Stockholm	301 ns
binary 984 99	Pajala	141 ns

Table 2: Linear search and binary search for when the zip code is in `Integer` format

linear 111 15	Stockholm	53 ns
linear 984 99	Pajala	11318 ns
binary 111 15	Stockholm	126 ns
binary 984 99	Pajala	136 ns

Table 3: Lookup for when the zip-codes are used as indexes

lookup 111 15	Stockholm	51 ns
lookup 984 99	Pajala	53 ns

Table 4: Lookup for when using Hash Buckets

lookup 111 15	Stockholm	158 ns
lookup 984 99	Pajala	69 ns

Table 5: Number of collisions when using Hash Buckets

Modulo	Unique	2 on same	3 on same	4 on same	Utilization
31327	8961	688	25	0	29%
28627	8878	770	26	0	31%
27773	8820	841	13	0	32%

Table 6: Lookup for ”Slightly better”

lookup 111 15	Stockholm	207 ns
lookup 984 99	Pajala	63 ns

Table 7: Steps in ”Slightly better”

Max Steps	59
Average steps when stepping	8.9
Average numb. of times we need to step	1.1

4 Discussion

Comparing table 1 to table 2 we see that the linear search is almost three times as fast when we compare integers as opposed to strings, and that even the binary search is faster. The reason for this is most likely in how `.equals()` and `.compareTo()` works. For strings the latter of the two compares them lexicographically on the basis of the unicode value of each character in the string. This means that it need to loop through each character in the string which will add time. The same goes for the `.equals()` method. It compares each character in the two strings and only if they are all equal will it return true. This means it will have to loop which will add time. Checking and comparing integers on the other hand does not require any looping, instead they simple need to compare the value filed of the integers, which is a constant time operation.

As seen in table 3 this lookup method is much faster than either of the previous two. It is actually a constant time operation The problem here though is that we're only utilizing 10% of the array we're storing the values in. If memory isn't a problem and they keys in question that we want to store aren't to large then this method could be a good choice. However it is more than likely that a better choice is to use a hash map to shrink the amount of wasted space and thus also being able to shrink the array size.

For the first of the two hash implementations we see in table 4 that the time require to lookup an entry is still really fast. It is actually a constant time operation for all entries except those where we had a collision. In those cases it is a $\mathcal{O}(c)$ operation where c is the number of collision at that spot. This means that trying to get the collisions down to as low as possible and also trying to ensure that we don't have more than one collision at those places where we get a collisions is really important, since this will directly affect the average time complexity of the lookup method.

In our implementation we went with using the prime number $31,327$ as our modulo operator to hash the keys. This because it gave us the least collisions while not lowering the utilized space of the array to much. Other numbers gave us a higher array utilization but at the cost of having more collisions which would require more linked nodes in our array. Since modern computers come with 8 gb of ram or more, a lower utilization of an array of this magnitude felt like an okay tradeoff for a faster lookup time.

We see in table 6 that the second hash implementation also has a constant time operation for the times where we don't have a collision. However since we handle collisions differently here we might get a hash index where nothing else has hashed too, but there was a collision on the index before thereby occupying this hashed index too. This will make it so that we need to place this hashed index one step further adding to the lookup time. So for indexes with collisions we will not have a constant lookup time but instead a time complexity of $\mathcal{O}(s)$, where s is the number of steps we need to go from where we started. Because of this it is important that the hash function distributes the keys as evenly across the array as possible, and with as few collisions as possible.

In table 7 we see that for our choice of modulo number we get a maximum step of 59, and that we on average need to step 9 times when we need to step, but that we on average only need to step 1.1 times when looking up every key. This indicates that our choice of modulo operator might have gotten the number of collisions down, but that it probably isn't that evenly distributed over the array.

All the code can be found here: GitHub