Hash Table

Laasyasree Desu

Fall 2023

1 Introduction

This assignment involved the implementation of different ”look up methods” for a certain type of the data. The data in question was list of zip codes with corresponding names of the areas and population sizes. The different methods implemented include linear search, binary search, using the key (zip code) as indices and implementing hash function.

2 Linear and Binary search method

The first task of the assignment involved implementing a linear search and a binary search on the list of zip codes. Before the implementation, the first step involved reading the data file and adding them to an array. This was done through implementing sections of code that was provided before hand.

2.1 Linear

The look up was implemented by using a simple for loop that would run as many times as the size of the array. An if statement would then check if the zip code being searched for was equal to the zip code of the current loop, if true the method returned the zip code. Otherwise the loop continued.

```
for (int i =0: i<arr.length; i++) {
 if (data.code.equals(arr[i])
    return data;
}
```

2.2 Binary search

The second search method, binary search took advantage of the fact that the array is already sorted by starting in the middle of the array and moving a quarter of the way forward or backward while keeping track of the first and last elements. The key (zip code being searched for) was compared to the middle value and if it was not found, the search field was modified accordingly.

```
if (arr[mid].code.equals(data))
   return arr[mid];
if (arr[mid].code < data)
   left = mid + 1;
else {
   right = mid - 1;
   }
```

2.3 Modification

Lastly, the two methods were modified to take the data as type Integer instead of strings. A new array was also created to add data as Integers.

2.4 Results

Zip code	Linear	Binary
11115	630	810
98499	17340	790

Table 1: Average time taken to search for zip code using linear and binary search

Zip code	Linear	Binary
11115	120	160
98499	6730	170

Table 2: Average time taken to search for zip code using linear and binary search

The results above (ref. table and table 2) show the time taken to search for specific zip codes using binary search and linear search. In the table 1 the binary search runt-time was significantly less than the linear search and remained relatively the same for both zip codes, while the linear search increased significantly for the second zip code, which can be attributed to the zip code being at the end of the array (the entire would need to be traversed). Table 2 shows the run-time after changing the type to Integer, which improved the overall run-time for both methods. This could attributed to the time saved by not having to change from strings to Integers.

3 Using key as index

The second task involved modifying the search method by taking the zip codes as indices of the array. Since the zip codes range up to 100 000, a new array of that size was created. Since the zip code being looked for is in the index of the same number, the "look-up" method was simplified to just an if statement that would check if there is a number present in that specific index.

3.1 Results

Array size	Time taken (ns)
11115	40
98499	40

Table 3: Average time taken to search with the zip code as index

The results above (ref. table 3) show the run-time for searching for zip code with the zip code itself being the index in the array. This gives a constant run-time since the specific index is directly accessed instead of having to traverse the array. This can be seen in the results as the run-time to search for the zip codes improved significantly and remained constant.

4 Hash function

To minimise the space waste, the third task involved implementing a hash function. Therefore, instead of using a large array and zip codes as indices, the zip codes were mapped to smaller indices of a specific range which would yield a smaller array. This was done by taking the "key" modulo some value "n" to give the index. The results of the calculations would be within the range of up to the value "n".

4.1 Collisions

The calculations using modular could however result in "collisions", where two or more keys are mapped to the same index. Therefore the third task also involved counting the number of collisions that would occur for different values of "n".

```
if (arr[index] != null)
    coll[index]++;
```

4.2 Results

Array size	Time taken (ns)
10000	5093
12345	2579
20000	3270
30000	2293

Table 4: Number of collisions for different values of n

The results below (ref. table 4) show the different number of collisions occurring for different values of "n". It can be observed that the number of collisions decreased with the choice of a more specific/unique number such as 12345 compared to 10000. It can also be observed that the collisions relatively decreased as the the value of n increased, however this would lead to a larger array and potentially become a waste of space.

5 Handling collisions - Array of buckets

To be able to handle the collisions, the next task involved creating and implementing an array of "buckets". This was done by creating an array of a certain size where the zip codes would be placed, however this time if a collision occurs the array index in question points to a linked list where the zip codes with the same modulo results would be stored. This meant that if a zip codes was being searched for, it would be directed to the corresponding array and index and traverse the particular linked list associated with it.

5.1 Improvement

A further improvement to the method was to handle everything in the same array, eliminating the need for the linked lists. The method would proceed as normal by calculating the modulo to get the index, however if a collision occurred it would instead traverse the array to find an empty spot. Therefore the array in question also needs to be large enough based on the size of the data.

```
while (arr[index] != null) {
   if (arr[index].code == data) {
       return arr[index];
```

5.2 Results

Value (n)	max steps	time taken to lookup
11115	362	170
98499	793	210

Table 5: Maximum amount of steps and average time taken to lookup zip code in array

Since all the values are stored on the array the look up method would give a a constant time complexity if the zip code being searched had no collisions. Otherwise the run-time would depend the number of ”steps” or places in the array that would need to be searched before finding the desired zip code. This can be seen in the results (ref. table 5) as the run-time is relatively constant for the lookup.