Hash Tables

Mahdi Nazari

October 2022

Introduction

In this assignment a new data structure will be presented, namely hash tables. This data structure will be implemented in two different ways and pros and cons of each will be described at the end. To get started we will begin with a table of zip codes which are stored in order in an array. Some benchmarks will be performed to see the differences between execution time of a linear searching and a binary search. Since the zip codes are of integer data type a new larger array will be implemented where the zip code values will be the index of that zip code in the array and some benchmarks will be performed to compare the execution time of searching an element in this larger array and the binary search in the previous method.

A table of zip codes

The first task is to store a table of zip codes in an array of the same size and each element in the array will be a node with properties string code, string name and integer population. A linear search method have been implemented which goes through the elements of the array one by one to find the searched element and then return the name of that element. Since the zip codes are stored in order in the array a binary search method can be implemented too. To compare each element in these searching methods the operation equals will be used. A benchmark will be performed to see the result of searching a zip code and the difference between execution time in a linear search method and a binary search method. The result is as the table below:

111 15 L	**994 99 L**	**111 15 B**	**994 99 B**
Stockholm (41 ns)	”Not found” (34791 ns)	Stockholm (1875 ns)	”Not found” (416 ns)

Table 1: The result of searching a zip code by linear(L)- and binary(B) search methods

As it sees in the table 1 the linear search method finds the zip code "111 15" faster since it is at the beginning of the array otherwise the binary search method is more time efficient to search an element by.
A slightly different type of class Zip was implemented where all zip codes were converted to integers before storing them in the array. Now the same benchmark was performed to see the difference in the execution time.

111 15 L	**994 99 L**	**111 15 B**	**994 99 B**
Stockholm (41 ns)	"Not found" (11750 ns)	Stockholm (1458 ns)	"Not found" (375 ns)

Table 2: The result of searching a zip code by linear(L)- and binary(B) search methods

The execution time was improved and this is because java performs the operation equals and compare.to faster on integers data type than string data type.

Key as index

Since the zip code is of integer data type a new class Zip was implemented where each zip code was stored at the same index as the integer zip code. This method requires to have a larger array and since the largest index can be 999 99 an array of size 100 000 was created to store the zip codes. Search method in this case is pretty simple to implement and that is just one raw code which will return the name of the node that is stored at the same index as the value of zip code. A benchmark was performed to compare the execution time between binary search in previous task and search method in this task.

111 15 B	**994 99 B**	**111 15 C**	**994 99 C**
Stockholm (1250 ns)	"Not found" (416 ns)	Stockholm (41 ns)	"Not found" (41 ns)

Table 3: The result of searching a zip code by binary(B)- and constant(C) search methods

As it was expected the execution time of the new search method is constant since the zip code gives a index to the array where this zip code is stored and program can have access to that place in constant time. This implementation is much time efficient but the largest drawback of this method is the waste of the memory. We are barely using 10% of the array that we created. To have an implementation that is both time and memory efficient we need to present a new data structure which is called hash tables.

Hash function

To solve the memory inefficiency in the previous task we need a function that gives us a fairly unique index for each zip code then we can avoid to create a very large array. This new function is called hash function and can simply be implemented by a modulo operation. This means we need to construct a hash function or rather find a number m that gives us a fairly unique index from 0 to m-1 for each zip code. We will soon find out that in some cases there will be more than one zip code which will have the same hash value and this is called collision. The aim is to find a number m which gives us the fewer number of collisions. To find the best number m, a number of tests were performed.

The test was as follows: a randomly number m was chosen between 10000 - 20000 and for each number m we run a program that would take modulo by m of each zip code in the zip code file and would give us a statistic of how many collisions will occur.

number m	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
11000	5024	2425	1198	607	290	99	30	1	0	0
10679	6549	2499	562	62	2	0	0	0	0	0
12000	5157	2397	1200	541	256	93	29	1	0	0
11587	7097	2216	343	18	0	0	0	0	0	0
15000	5536	2366	1099	474	178	21	0	0	0	0
14449	7122	2117	384	47	4	0	0	0	0	0
20000	6404	2223	753	244	50	0	0	0	0	0
19991	7712	1653	282	27	0	0	0	0	0	0

Table 4: A randomly chosen number m and the collisions that occur by taking zip code modulo m

The table 4 demonstrates that the larger number m gives fewer collisions and also a primitive number m gives fewer collisions compared to a larger non-primitive number m. Choosing a number m is trade off between the number of collisions and the array size.
Now we can construct a hash function with a suitable number m, then we still need to handle the collisions. We can handle collisions in two ways, namely by a hash table with bucket and hash table without bucket.

Hash table with bucket array

One way to handle a collision is by an array of buckets and each bucket is a list of elements that have the same hash value. When we insert an element

(zip code in this case) first we get an index by hash function then we need to check if there is already an element in that bucket if so we continue in that bucket until we reach the end and then we can insert the new element in that position. To search an element we need do the same we take the hash value of the searched element and then we get an index, we jump to that index and will check if there is any element there or not, if not then we can just return and the searched element is not existing in the bucket array. Otherwise we need to check if the element has the same zip code as the searched element, if it has then we can return the name of the node. The insert function is as follows:

```
private void insert(Node newNode){
        int index = (newNode.code \% mod);    // hash function
        Node nxt = bucket[index];
        Node prev = null;
        if(nxt == null){
            bucket[index] = newNode;
            return;}
        while(nxt != null){
            prev = nxt;
            nxt = nxt.next; }
        prev.next = newNode;
        return; }
```

Hash table with array

Another solution for handling a collision is using a usual array without any bucket. To insert an element we take the hash value of that element and then we jump to that index if there is already an element in that index we continue jump to the next position in the array by incrementing the index. We will stop incrementing as soon as we find an empty slot that we can insert the new element. To avoid falling out of the array edge we take module by the size of array every time we increment the index. Having a tight array increases the risk of going through many elements to find an empty slot for inserting a new element. The insert function is as follows:

```
private void insert(Node newNode){
        int index = (newNode.code \% mod);
        Node nxt = bucket[index];
        if(nxt == null){ bucket[index\%mod] = newNode;
            return;}
        while(bucket[index\%mod] != null){
           index++; }
        bucket[index\%mod] = newNode;
```

```
return;
}
```

To search an element we need to go through the same procedure. We take the hash value of searched element and jump to that index and if there is no element there we can just return and the searched element does not exist in the array. Otherwise we check if the element has the same value as the searched element if so we have found the searched element. Otherwise we continue in the array by jumping to the next index and compare the value of that element with searched element. This procedure continues until we find the searches element.

Hash table with bucket array vs hash table with array

A benchmark was set so to get a statistic of how many element does each of these methods goes through until it finds the searched element. A global variable "comparison" was declared in each of these methods. The variable comparison will be incremented every time the program compares the value of the searched element with an element in the bucket array or array.

Size	**53151(1)**	**44696(1)**	**70224(1)**	**53151(2)**	**44696(2)**	**70224(2)**
	Lidköping	Hålanda	Örebro	Lidköping	Hålanda	Örebro
100067	3	2	1	19	6	1
11587	1	1	1	13	1	4
15331	1	1	3	5	1	10
21269	1	1	1	1	1	1

Table 5: (1) hash table with bucket array, (2) hash table with array. The sizes are chosen as prime numbers

The result shows that the hash table with a bucket goes through fewer elements until it finds the searched element. Since in hash table with an array the elements with the same hash values are stored in raw close to each other, this increase the risk that those elements that have a unique hash value have to go through many elements to find an empty slot to get inserted. Searching method goes through the same procedure and therefore a hash table with an array has to go through more elements compared to a hash table with a bucket array. As it has been mentioned earlier if the size of array increase the risk for collision decrease and thus program has to go through fewer elements to find the searched element which is confirmed in the table 5.