Hash Tables

Ayda Ghalkhanbaz

Fall 2023

Introduction

This assignment involves implementing a data structure called hash tables. Hash tables are a common and efficient way for data storage and retrieval. They use arrays to store data and keys to access the data in a relatively constant time. Throughout this assignment, we will work with a file containing information about cities and their zip codes. We'll start by implementing inefficient solutions for data retrieval and gradually find our way to a more optimised solution. Therefore, this assignment includes multiple smaller tasks and benchmarking. The results are shown under each section and will be discussed later in the report.

String vs Integer Zip Code

The skeleton of this assignment was already given in the instructions. It includes tasks such as separating every line after comma and defining attributes such as `code`, `name` and `population` for each line of the given file. These data are then stored in an array of size 10000. The primary purpose of this task is to compare the execution time when looking up a zip code in the array with a focus on whether the zip code is stored as a string or an integer.

Two lookup methods, linear and binary, were implemented for both versions (string and integer). The linear lookup has a simple structure. It iterates over the array and checks if the zip code of the current node in the array is the one we are looking for. The comparison is done using `.equals()`. As it is discussed in the previous assignments, this linear searching method has a time complexity of $\mathcal{O}(n)$.

Since the data we stored in the array was sorted, we were able to use binary lookup as well. This method has similar structure and function as it was in the previous assignments, that is, we navigate to the middle of the array and check whether the code we seek is greater or lower than the middle node. By consistently checking only half of the array, this process

continues until the target zip code is found. The time complexity that we gain by employing this method is $\mathcal{O}(\log(n)$.

Result

The execution time of both approaches for linear and binary lookup was measured and the results are presented in table 1.

Zip code	String version	Integer version
linear(11115)	20	17
linear(98399)	33000	11600
binary(11115)	297	236
binary(98399)	191	165

Table 1: Execution time of linear vs binary lookup with both String and Integer versions in ns

Hash Table: Key as Index

Using the zip codes as index was the next task in the assignment. Implementing this will of course buy us more time for lookup because accessing an element in an array has a time complexity of $\mathcal{O}(1)$ which is much faster than the previous task. However, implementing this costs a lot of memory, because we start populating the array at index 11115 (the first zip code is 11115) to index 98499 (the last zip code is 98499), which means that we need an array of size 100000.

Hash function allows us to use a compact array and still access the data using indexes. There were two main changes in the implementation for this task. First, when populating the array, each node is assigned an appropriate index using a hash function. The hash function used in this assignment is very simple and it only utilises a modulo operation. The second change was the `lookup()` method which became more simple and contains only one line of code as we use hash function in order to find the index of the sought-after node. The code snippet below illustrates the `lookup()` method for this task.

```
public String lookup(Integer zip){
    // zip%mod is the hash function. mod = array size
    return data[zip%mod].name;
}
```

Result

In this task the number of collisions was calculated for an array of varying sizes. The results are showcased in table 2.

Modulo/size	Unique	2 collisions	3 collisions	4 collisions
10000	4465	2414	1285	740
13513	7387	1976	299	12
20000	5404	2223	753	244
28447	8648	990	36	0
31327	8691	688	25	0

Table 2: Number of collisions in arrays with varying sizes (PS! the bold numbers are prime)

Handling Collisions

To minimise the number of collisions, two different approaches were implemented, and each of them will be discussed shortly.

Buckets

The first implementation for handling collisions involved a minor modification to the `Node` class structure. I added a node attribute called `next`, which comes handy while adding nodes with the same hash value which causes collisions.

In the `add()` method, instead of just adding a node into the hashed index, we first check whether that cell in the array is empty or not. If the cell is empty, then we can place the node there. However if the cell is not empty, it means there is another node sitting there with the same hashed key. In this case, we simply set the `next` pointer of the current node to point to the new node. Thus, if there are collisions within a cell of the array, instead of only one node, we have a linked list of nodes pointing to each other.

In a case of no collisions within a cell, the `lookup()` method simply returns the name of the target city. But if there are collisions, then we can find the city using a `while` loop iterating through the list of nodes with the same hashed key until we find the desired city. The code snippet below illustrates the implementation of the `lookup()` method for this task.

```
public String lookup(Integer zip){
    Integer hashKey = zip%mod;
```

```
        if(data[hashKey].code.equals(zip)){
            return data[hashKey].name;
        }else{
            Node curr = data[hashKey];
            while(curr != null){
                if(curr.code.equals(zip)){
                    return curr.name;
                }
                curr = curr.next;
            }
            return null;
        }
    }
```

Open Addressing

The second solution to handle collisions is open addressing. In this approach, when collisions occur within a cell in the array, instead of creating a bucket of nodes within the same cell, add the new node into the next nearest empty cell in the array.

The `add()` method implemented for this task has a simple logic, that is, iterating over the array to find the nearest empty cell to the hashed index in case of collisions. The only problem was that we could go beyond the boundaries of the array. This was solved by wrapping around the hashed index. Below you'll find the implemented `add()` method.

```
    public void add(Integer code, Node n){
        Integer hashKey = code % mod;
        while(data[hashKey] != null){
            // wrap around
            hashKey = (hashKey + 1) % mod;
        }
        data[hashKey] = n;
    }
```

The `lookup()` method contains a `while` loop, iterating over the array until it finds the target city. The loop is maintained within the array using the wrap-around technique. The following code snippet shows the implementation for the `lookup()` method.

```
public String lookup(Integer zip){
        Integer hashKey = zip%mod;
        while(data[hashKey] != null &&
                              !(data[hashKey].code.equals(zip))){
            // wrap around
            hashKey = (hashKey + 1) % mod;
        }
        return data[hashKey].name;
    }
```

Result

For this task, the number of nodes to be visited before finding the target node was calculated for three randomly selected cities. The results are shown in table 3 and 4.

Zip code	**Modulo/array size**			
	10000	**13513**	**20000**	**28447**
931 93	4	0	3	0
461 44	1	0	1	0
514 70	1	0	0	0

Table 3: The number of nodes to be visited before finding the target in bucket implementation

Zip code	**Modulo/array size**			
	10000	**13513**	**20000**	**28447**
931 93	5871	5	217	0
461 44	34	0	24	0
514 70	645	0	15	0

Table 4: The number of nodes to be visited before finding the target in open addressing implementation

Discussion

From table 1, we can observe that the lookup execution time for integers is faster than that for strings. Although both approaches utilise the same implementation for linear and binary search, the difference in their execution

time has its roots in the comparison operations. Comparing strings is an operation that is done character by character either through `.equals()` or `.compareTo()`. These operations involve loops iterating through the strings which lead to more execution time. However, integer comparisons are done by comparing the numeric value of the integers due to their nature, which will buy us some time.

The main purpose of compiling the results in table 2, was to show how important it is to choose an appropriate modulo number in hash functions. As the first observation, we can see that as the size of the arrays grows, the number of collisions decreases. This is due to more space allocated to data which leads to less number of collisions.

However, larger array size isn't the only optimisation for having a low collision rate. As we can see in the table, prime numbers are also good candidates and they relatively give less collisions. There are many reasons why prime numbers are an optimisation, one reason could be that they reduce the chance of having common factors with other numbers. Also, prime numbers, larger numbers in particular, tend to offer a wider range of remainders which leads to a more even distribution of keys within the array.

From tables 3 and 4, we can observe that in case of collisions, the bucket implementation in general visits fewer nodes compared to the open addressing implementation. Although both approaches have the same time complexity, which is $\mathcal{O}(1)$, open addressing might have a slower execution time. The reason lies behind their implementation logic.

When collisions occur, the bucket implementation needs to iterate through a linked list with a few nodes before it reaches the target node, while in the open addressing implementation, the number of nodes to be visited before we reach the target may vary and it actually can be larger. This is because it is a little unpredictable how many nodes should be visited first. Thus, their time complexity depends on how many nodes from the hashed key need to be visited before landing on the target node.