Hash Tables

Abdinasir Abdi

October 2024

Introduction

This report concludes the eight assignment in course ID1021, taken in the fall of 2024. The report explores different implementations of accessing and storing larger data sets and their benefits and drawbacks.

Perceptive Construction

In the first task of the assignment, information is extracted from a given file and stored as sequential nodes in an array. The file is in CSV format (Comma-Separated Value), and every item is inserted in an array of entries. All entries consist of the code address, name, and the area's population. Linear and binary search were then used to search for different zip codes as strings in individual nodes. A benchmark for each search method was made and the following results were obtained:

Iterations	**Target Zip**	**Linear Search**	**Binary Search**
1000000	11115	54ns	44ns
1000000	98499	$12\mu s$	46ns

Table 1: Benchmark Results for Linear and Binary Search for Different Zips

Linear search traverses through the array sequentially and therefore zip code **98499** takes longer to locate. Finding both zip codes for using binary search takes almost the same time.

In the second part of the task the zip codes, which are stored as strings, are converted to Integers before creating the entries. The same searching algorithms were used as previously and the following results were obtained from the benchmark:

Iterations	Target Zip	Linear Search	Binary Search
1000000	11115	25ns	43ns
1000000	98499	8.9μs	44ns

Table 2: Benchmark Results for Linear and Binary Search for Different Zips

Linear search performed much better by having the zip codes be stored as integers. Comparing strings, in our case numeric strings, the comparison is done character by character. This makes it more complex it involves checking the character encoding and also handling the string lengths. On the other hand, comparisons between integers are easier because they involve direct numeric comparisons. This is handled at a lower level in the processor and usually performs constant time $O(1)$. The difference in performance for binary search is minimal.

Key as Index

The previous search algorithms become significantly less efficient as the data sets grow. In the next step of the assignment, a different approach was used. The integer zip code is used as an index in an array and each zip code is stored at the equivalent array position. The only things that changed are the size of the data array(hundred thousand) and how the array is populated. A *Lookup* method was then implemented. The following results were obtained using a benchmark over the lookup method:

Iterations	Target Zip	Lookup
1000000	11115	45ns
1000000	98499	45ns

Table 3: Benchmark Results for Lookup Searching for Different Zips

The result confirms the access time complexity to be constant time $O(1)$. The lookup method utilizes an array indexed by the zip codes and maps them directly to array indices. This eliminates the need to traverse(linear search) sequentially and does not require iterative comparisons, making it highly efficient.

The disadvantage of this approach is that it consumes much more memory than necessary. Even though the file contains less than a thousand zip codes, memory for a hundred thousand elements is allocated for this test. One way to address this is by using a hash function. It allows one to reduce the total memory needed by compressing zip codes into a smaller array by reconstructing the zip code(key) into an index. An example would be to

store the zip codes at an index equivalent to an exact modulo of each zip code.

Modulo\Collisons	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
10000	1345	3665	8182	1122	406	203	104	48	88
20000	5402	6225	753	9221	20	0	0	0	0
12345	1522	2149	345	63	18	0	0	0	0
54321	9385	452	0	0	0	0	0	0	0

Table 4: Number of collisions for hashing the given zips

In this approach, multiple zip codes may end up with the same equivalent index, resulting in collisions between zip codes. A data array called **bucket** is implemented to manage further and minimize collision.

```
public ZipKeyWithBuckets(String file, int bucketSize) {
    postnr = new ArrayList[bucketSize];

    try (BufferedReader br = new BufferedReader(new FileReader(file))) {
        String line;
        while ((line = br.readLine()) != null) {
        String[] row = line.split(",");
         int code = Integer.valueOf(row[0].replaceAll("\\s", ""));
      int bucketIndex = code % bucketSize;

    if (postnr[bucketIndex] == null) {
     postnr[bucketIndex] = new ArrayList<>();
            }

    postnr[bucketIndex].add(new Area(code,row[1], Integer.valueOf(row[2])));
            }
    }

    catch (Exception e) {
        System.out.println("File " + file + " not found");
    }
}
```

In the new implementation, the program handles collisions using a method called chaining. Chaining involves handling collisions using a hash table. When multiple keys obtain the same hash index the values are typically stored in a linked list data structure, instead of overwriting the existing value at that position. For the program above a *ArrayList* implementation was used. In the hash table, each index points to a bucket containing multiple

entries. Whenever a new entry is added, the program checks the bucket for collisions and appends the new data if necessary. This way minimizing the bucket to store the zip code and the associated data.

The hash function used in this implementation is simply a modulo operation that maps the zip codes to each specific bucket within the array. When then performing the lookup method the program uses the hash function to determine the correct bucket.

```
public Area lookup(String zip) {
    int zipCode = Integer.valueOf(zip.trim());
    int bucketIndex = zipCode % postnr.length;

    if (postnr[bucketIndex] != null) {

        for (Area area : postnr[bucketIndex]) {
            if (area.code == zipCode) {
                return area;
            }
        }
    }
    return null;
}
```

Before the performance is measured, the program is slightly improved by using *open addressing*, also known as *linear probing*, to handle collisions. This interchange handles it by storing elements directly in the main array therefore no longer a need for separate buckets. When a collision occurs at the hash index it traverses forward through the array looking for the next available empty position, instead of creating a new bucket. The lookup method also needs to be refined but the process is similar. It starts from the hashed index and checks if the element at that position is equal to the desired value. If true, it returns that value, and if not it sequentially traverses through the array checking each subsequent slot until the target value is found or an empty slot. If the search ends on an empty slot it means that the element does not exist and the program returns null.

Modulo	**11115**	**98499**
10000	2	1400
11000	3	300
12000	1	500
13000	1	12
14000	2	180
15000	1	10
Average Lookups	1.67	290

Table 5: Number of lookups for zips 11115 and 98499, along with average lookups across all zips

The bucket-based(chaining) implementation is easier to manage because each index can accommodate multiple entries, making it simpler to handle collisions. Bucket can be implemented with data structures like linked lists, allowing for dynamic resizing and efficient insertion and deletion. The drawback with this is that additional memory is needed to store references for the linked lists, this increases memory usage. *Open addressing* on the other hand is better for memory efficiency. The elements are stored within the hash table, eliminating the need for linked lists. The biggest drawback of this implementation is as the table fills, search and insertion performance increases substantially. Both implementations work well and are efficient and depending on use, one becomes better than the other.