Hash Tables

William Frid

Fall 2023

Introduction

When storing data there a different structures one can choose from. One of which is hash tables. Hash tables build on the principle of hashing the indexes and then grouping them in an array for instance close to the index each to the hashed one. Then upon search, the searched-for index is hashed, then that hash is used to quickly navigate closer to the correct element, followed by some final comparisons.

This report will explore hash tables with and without buckets (also known as open addressing), and compare these to each other and linear search to show their different pros and cons. To do this, Swedish ZIP codes stored in a CSV-file will be used. For hashing, different modulo will be used and by doing so, also leads to the exploration of of different modulo for hashing behaves alongside the size of the hash table.

Searching Trough ZIP codes

When searching through an array of Swedish ZIP codes, one can take a few different approaches. Since the data in the CSV file is sorted, we will explore two options that will vary drastically: linear and binary search. Starting with the linear search which will simply traverse through the array until the desired array is found, and lastly, a binary search using recursion. The snippet is seen below with a self-explanatory code. *Note how we're comparing strings and not integers at this stage.*

```
Node binarySearch (String zip) {
    return binarySearch(zip, 0, max);
}
Node binarySearch (String zip, int start, int end) {
    if (start == end && data[start].code.compareTo(zip) != 0)
        return null;

    int mid_index = start + ((end - start) / 2);
```

```
    Node mid = data[mid_index];

    if (mid.code.compareTo(zip) == 0)
        return mid;
    else if (mid.code.compareTo(zip) > 0)
        return binarySearch(zip, start, mid_index - 1);
    else
        return binarySearch(zip, mid_index + 1, end);
}
```

Based on common knowledge about these two types of searches, one can expect that the binary search is generally faster thanks to its time complexity $O(log(n))$ versus the linear search's $O(n)$. However, the linear search should in theory be after if the ZIP code searched for is close enough to the beginning since it requires less logic and use of the stack. This was tested using a benchmark whose results are displayed in Table 1. Take into consideration that the ZIP code 111 15 is the first element and 984 99 is the last element in the array.

ZIP	**Linear (ns)**	**Binary (ns)**	**Ratio**
111 15	80	250	0.30
984 99	33000	300	111.00

Table 1: Benchmark comparing two different scenarios.

But can these results perhaps be improved? Some of the methods were modified to treat the ZIP codes as integers instead of strings. The result of this benchmark is seen in Table 2 which tells us that comparing strings is slower than comparing integers with the linear search gaining the most improvement. This could be because of the simpler type of data structure which leads to faster comparisons.

ZIP	**Linear (ns)**	**Binary (ns)**	**Ratio**
111 15	70	100	0.70
984 99	12000	150	81.00

Table 2: Benchmark comparing two different scenarios.

Key As Index

To improve these results even more, the array containing all nodes with ZIP codes was modified so that each ZIP code was located at that exact index. Meaning that the array size was increased to 100,000. Then a method called

`lookup()` was added which took a string as an argument. The string held a zip code which the method converted to used as an index for navigating the array. This setup was too benchmarked with its results shown in Table 3. We see that it is fast, must not as fast as binary search. We can also see that the results for the different elements are the same. The equal results are thanks to the lookup algorithm and the speed increase (compared to binary) due to the immense size of the array which is 90% empty.

ZIP	**Lookup (ns)**
11115	160
98499	160

Table 3: Benchmark showing the lookup method.

One way of handling this size issue is by creating an additional array that holds hashed keys. That is an array without gaps. To do this, the first step was to find a module to use that would hashify the keys without creating too many collisions (multiple keys leading to the same ZIP code leaving some ZIP codes without a key). Note that because of the way modulo works, a larger doesn't mean less collision. In this specific case, $mod10000$ leads to less collisions than $mod12345$. So to use modulo, one has to find the best number that balances the collisions and size of the array to make it worth using.

Handling Collisions

To handle collisions, one can use hash tables. The hash table that was used for this report was an array containing node arrays (each node representing a ZIP code). The hash table array wasn't strictly a 2-dimensional array since the inner arrays, called a "bucket", were initialized to length 1. At this stake, the hash table consisted of n many empty buckets. Then upon insertion, the program hashed each zip using $mod13513$ and then inserted the node in the bucket where at index was given by the hash. This led to different large buckets. Then using a lookup method, the program hashed the ZIP searched for, found the bucket, and then compared the few nodes there directly to the ZIP (non-hashed). The two methods used for adding to buckets and looking up data are shown in the snippet below.

```
void bucketAdd(Node n) {
    Integer hash = n.code % mod;
    if (hash_table[hash][0] == null) {
        hash_table[hash][0] = n;
        return;
    } else { // Expand bucket.
```

```
        Node[] newBucket = new Node[hash_table[hash].length + 1];
        int i = 0;
        while (i < hash_table[hash].length)
            newBucket[i] = hash_table[hash][i++];
        newBucket[i] = n;
        hash_table[hash] = newBucket;
    }
}

Node lookup (Integer zip) {
    Node[] bucket = hash_table[zip % mod];
    for (Node k : bucket)
        if (zip.compareTo(k.code) == 0)
            return k;
    return null;
}
```

To benchmark this algorithm one identical to the previous used was run. Table 4 containing the benchmark results shows an eight times improvement in speed. It is worth pointing out that using a linked list instead of an inner array can be an advantage thanks to not needing to allocate a new array upon insertion.

ZIP	Lookup (ns)
11115	30
98499	30

Table 4: Benchmark showing the lookup method using a hash table.

Improving The Algorithm (Open Addressing)

The last thing that was tested was open addressing. This is the concept of only using a larger array as a hash table without buckets and instead using null elements in the array to separate hashes. This however is tricky since one must choose a mod and array size that works well together to get just enough null space in the array. Several benchmarks were run on this testing searching for different ZIP codes using different mods. In Table 5 we see a comparison of $mod13513$ and $mod19319$ with an array size of 20000. It tells us that in some cases, open addressing can in some cases give a better performance, but I generally slower overall (ZIP 98499 for instance).

ZIP	Mod 13513 (ns)	Mod 19319 (ns)
111 15	30	30
227 35	30	30
352 46	40	30
441 91	60	30
591 62	50	30
694 60	190	40
737 91	140	40
984 99	110	360

Table 5: Benchmark comparing mods for lookup.

Discussion

The benchmarks run for this report are wary in precision. Even though they all were run with extremely high amounts of tries (some up to two million attempts), not all of them acted 1000 times then decided the total sum by 1000. This is a good thing to do due to in bad precision default Java methods have for benchmarking. In addition, other types of graphs could have been added for better visualisation, especially for comparing speeds between the two types of hash tables (with and without buckets).

Conclusion

A hash table with buckets is a powerful data structure that offers great speeds for both insertion and searching (lookup). This comes with the drawback of a larger amount of code needed and the work of experimenting to find the ideal modulo if that is the way the hashing will be carried out.