10. Hash table

Kristiāns Vinters

Fall 2023

Introduction

I solved the assignment in Go. I used Go because I want to become more familiar with it. Source code and benchmark data is available on GitHub[*].

Implementation

There weren't any particular difficulties in implementing a hash table (hash map) in Go for use with the `postnummer` dataset.

The hashmap was defined with a simple struct.

```
type NodeHashMap struct {
    buckets    [][]Node
    numBuckets int
}
```

The hash map is created by the `CreateHashMap()` function, which takes the parsed CSV data and number of buckets as arguments.

```
func CreateHashMap(data []Node, numBuckets int) *NodeHashMap {
    hashMap := NodeHashMap{
        buckets:    make([][]Node, numBuckets),
        numBuckets: numBuckets,
    }

    for _, node := range data {
        hash := node.CodeNum % numBuckets

        if hashMap.buckets[hash] == nil {
            hashMap.buckets[hash] = make([]Node, 0)
        }
```

[*]https://github.com/Phanty133/id1021/tree/master/10-hashmap

```
        hashMap.buckets[hash] = append(hashMap.buckets[hash], node)
    }

    return &hashMap
}
```

The lookup function takes a zip code as a string and returns a pointer to the node with the given zip code and the number of elements in the bucket, which is used for benchmarking. The function first calculates the hash, then finds the right entry in the resulting bucket. If the entry is not found, `nil` is returned.

```
func (data *NodeHashMap) Lookup(zip string) (*Node, int) {
    zipNum, err := ZipToNum(zip)
    // ... Error handling here ...

    hash := zipNum % data.numBuckets

    for _, node := range data.buckets[hash] {
        if node.Code == zip {
            return &node, len(data.buckets[hash])
        }
    }

    return nil, 0
}
```

Benchmarking

I performed benchmarks for `111 15` and `984 99` lookup for all implementations,as well as hash map lookup time vs bucket count and number of collisions on the `postnummer` dataset vs bucket count. All measurements were repeated 5000 times.

For analyzing run times, I measured the raw execution time for every repeat in nanoseconds, which I stored in an array and then wrote to a `.csv` file. I used LibreOffice Calc to calculate the mean, median, min, and max times and plot the graphs.

Implementation comparison

Fig. 1 shows the lookup times for the various implementations. For `111 15` all implementations are very fast, but for `984 99` the linear search is by far the slowest. The reason is because the linear search starts from index

0, which happens to be `111 15`, whereas `984 99` is on the other end of the dataset. Because of the small dataset, it's difficult to distinguish between the runtimes of the other algorithms. Overall, the fastest median time is for the integer binary search, but that's because the plain map and hash maps take a string parameter, so there's a conversion overhead. If the plain map and hash map were to take an integer parameter, they would be as fast as the binary search.

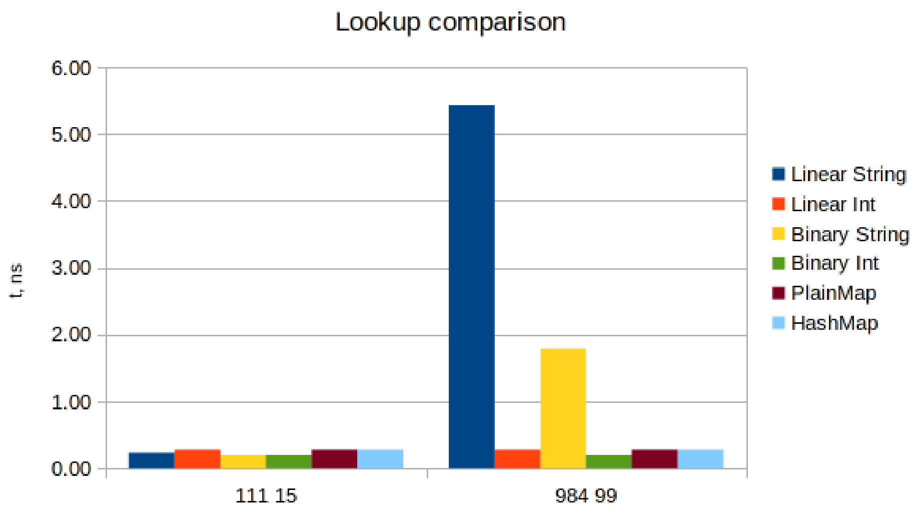

Figure 1: Lookup times for the various implementations.

Value	**Type**	**Algorithm**	**Median time, ns**
`111 15`	String	Linear search	0.23
		Binary search	0.28
		Plain map	0.28
		Hash map	0.28
	Integer	Linear search	0.20
		Binary search	0.20
`984 99`	String	Linear search	5.43
		Binary search	0.28
		Plain map	0.28
		Hash map	0.28
	Integer	Linear search	1.79
		Binary search	0.20

Figure 2: Implementation lookup median times

When comparing string and integer lookups, integer lookups are noticeably faster. The difference isn't as great for binary search ($0.08ns$), but for linear search there's a 3.0x performance improvement. This is because integer comparisons are significantly faster than string comparisons. When comparing strings, you need to compare each character, whereas integers can be compared in a single operation. The reason why there's a lesser improvement for binary search is because the linear search performs significantly more comparisons.

Bucket collisions

I measured the number of collisions in the `postnummer` dataset for every modulo from 1 to 100000 and plotted in fig. 3. The relationship is non-linear, with the number of collisions decreasing exponentially as the modulo increases.

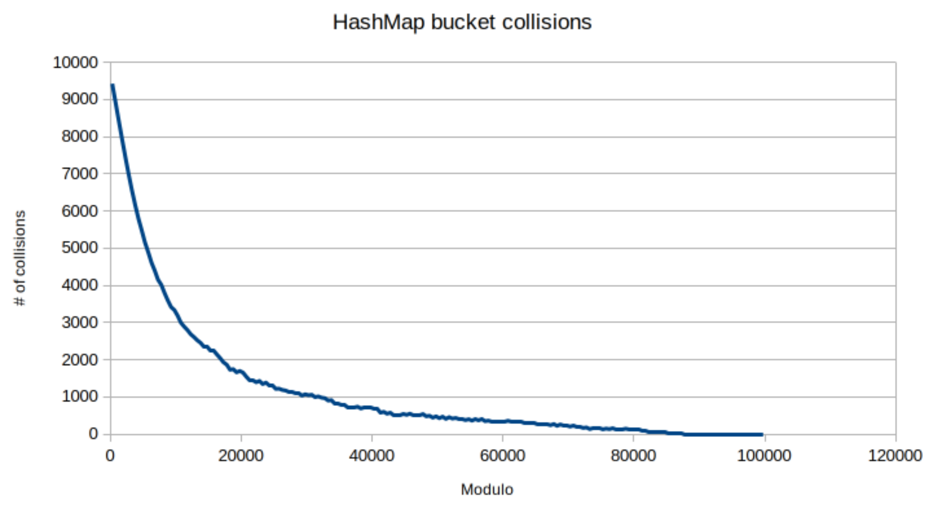

Figure 3: Bucket collisions vs modulo

Hash map lookup time vs bucket count

I benchmarked 5000 random, consistent lookups for hash map modulo 10, 100, 500, 1000, 2500, 5000, 7500, 10000, 15000, 25000, 37500, 50000, 62500, 75000, 100000. The results are shown in fig. 4. The lookup time increases as the modulo decreases, therefore mean bucket size increases. While the bucket size keeps decreasing after modulo 100, the lookup time doesn't decrease as much. Therefore, an optimal modulo for this particular dataset appears to be between 100 and 1000.

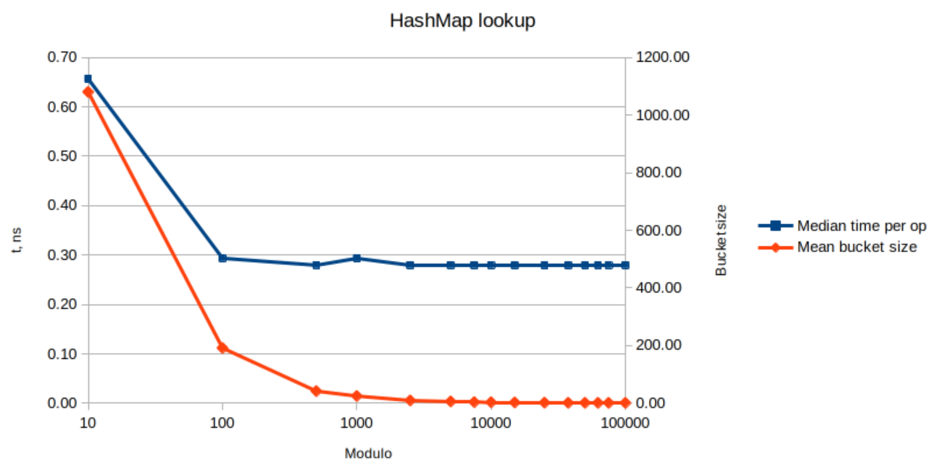

Figure 4: Hash map median lookup time and mean bucket size. Left Y axis is lookup time, right Y axis is bucket size.