Paper Assignments Lab 2

Mikolaj Gawrys

1 First assignment

The table below presents the performance of two hashing functions, $k \ mod \ 16$ and k / 16. In both cases, a HashTable of size 16 was populated with integer values from 0 to 39 inclusive. Code can be found in the appendix.

	$k \ mod \ 16$	k / 16
Avg. keys per slot	$2,5$	$13,33$
Load factor	2,5	2,5
Collisions	24	37

Table 1: Performance of given functions

1.1 Observations and conclusion

By analyzing the results above one can clearly see that the first hashing function ($k \ mod \ 16$) is far superior to the second function in all aspects. The average number of keys per slot increases by a factor of about 5 with the second hashing function, which translates to less uniform distribution of keys across the hash table. What is important to note here is that when calculating the average number of keys per slot, one does not take into consideration empty buckets [1].

The load factor is the same in both cases which is quite intuitive, since the hashtable itself is static - one knows the values that will be stored in

the hashtable and the size of the hashtable itself from the get-go (for this particular assignment), and those numbers are fixed. Hence, the load factor, expressed as

$$Load\,factor = n/m = 40/16 = 2,5 \tag{1}$$

where n is the number of elements stored in the hashtable and m is the number of buckets, is equal in both cases. What is important to note is that a load factor this high is not acceptable in most scenarios, since the trade-off between time and space costs is not preserved. In this particular case, this issue could be addressed by increasing the initial capacity of the hashtable (around 40, since we know how many values we will store), which would keep the load factor below 1 and imply a more uniform distribution.

The number of collisions also reflect the differences in the distribution of the keys across the hashtable between the two hashing functions. And, since the first hashing function gives less collisions, it is preferred.

In conclusion, the first hashing function ($k\ mod\ 16$) is the better choice for this particular problem. It allows for a more uniform distribution across the buckets, keeping the number of collisions as well as the average number of keys per slot lower.

2 Second assignment

Before exploring potential techniques to address this issue, it is important to acknowledge that the traversal time of a linked-list (which is $O(n)$) *itself* is hard to alter without using underlying data structures. For example, one might think that trying to mimic binary search (inserting an extra pointer to the middle) would reduce the search time, but $n/2$ is, by definition of Big-O, $O(n)$ still. Thus, the following techniques focus on improving this search time not by tweaking the linked-list itself, but by using underlying data structures (that use linked-lists in their implementation) with the usage of hash values.

2.1 Cuckoo hashing

One of such techniques is Cuckoo hashing [2]. It revolves around using two hash tables and two hashing functions, which reduce the probability of collisions. Cuckoo hashing is very simple to implement and provides $O(n)$ space complexity, worst-case *constant* lookup time, $O(1)$ deletion time as well as

a dynamic framework which can be adapted for specific use cases. However, several obstacles still remain. For example, it may require more rehashing operations than other algorithms, which is significantly more visible with higher load factors. The lack of full understanding of the underlying implications of this method and the constant search for improvement are yet to be addressed.

2.2 Bloom filters

Another useful technique is using a Bloom filter [3]. It is a probabilistic data structure that by using multiple hash functions to map keys is able to quickly determine whether a given element exists in a linked-list. This is because the filter is able to eliminate the keys that are not present, making the comparison time much shorter. It is characterised by fast search times ($O(n)$, where n is the number of hash functions used), very little memory required to store the hash values and false negatives ***only***. That means if the filter reports a given key is present, it is with 100% certainty. One of the biggest downsides is that the filter being a probabilistic data structure might produce false *positives*. It also uses much memory for low false positive rate. Meaning, if a certain threshold is to be achieved with regards to precision, the cost in memory might negate the benefits of this technique in the first place. Thus, there is a trade-off one must take into consideration when designing this algorithm.

2.3 Separate chaining

Lastly, quite an easy technique to implement, not an irrelevant one nevertheless, is separate chaining. For each hash value in our linked list a bucket in a hash table is created. Each of the buckets has a linked list as well, which contains the keys. That helps with search times, since smaller subsets need to be traversed. Keys that produce equal hash values are put into one linked-list, which is smaller than the initial one. This technique is the easiest to implement by far, and it may be a good solution when a good load factor is maintained. If it is about 0.7 and less an (about) $O(1)$ complexity for insertion, deletion and lookup is preserved. Otherwise, and in general, both the space and time complexity of this approach are $O(n)$.

2.4 Summary and conclusion

The three techniques: Cuckoo hashing, Bloom filters and separate chaining are all viable solutions to different sets of problems, depending on the requirements and/or expectations. Bloom filters, being a probabilistic data structure might provide solutions and insights unseen when using other techniques, at the same time being hardest to implement and requiring specific knowledge. Separate chaining is the easiest to implement by far, but loses in terms of both spatial and temporal complexity. Cuckoo hashing, being easy to implement, behaving better than separate chaining with large lists and having excellent lookup/deletion time seems to be a good trade-off between implementation difficulty and performance, which would be the author's preferred choice for this problem.

3 Third assignment

The statement, "The runtime of an algorithm A is ***at least*** $O(n^2)$" is meaningless because Big-O notation tells one about the *upper bound* runtime of a given algorithm. Hence, there is a contradiction embedded into this very statement - Big-O tells us the worst case scenario, when *at least* suggests that the worst case scenario is not actually the worst.

4 Fourth assignment

The formal definition of Big-O is as follows: A function $T(N)$ is $O(F(N))$ if for some constant c and for all values of N greater than some value n_0:

$$T(N) <= c * F(N) \tag{2}$$

4.1 Is $2^{(n+1)} = O(2^n)$?

We can prove this directly and work this example out by using the definition. We consider the inequality:

$$2^{(n+1)} <= c * 2^n \tag{3}$$

Which can be rewritten as:

$$2^n * 2 <= c * 2^n \tag{4}$$

Which becomes:

$$2 <= c \tag{5}$$

By choosing $c = 4$, one can clearly see this inequality holds for all n_0 chosen, hence the answer is **yes**.

4.2 Is $2^{2n} = O(2^n)$?

Consider the inequality:

$$2^{2n} <= c * 2^n \tag{6}$$

We can divide both sides by 2^n which gives, by the property of powers:

$$2^n <= c \tag{7}$$

Clearly, there is no constant c, that for a big enough N, bounds 2^n (eg. $c = 2$, $n_0 = 3$). Ergo, the answer is **no**.

5 Fifth assignment

1. $O(n)$
2. $O(n)$
3. $O(n^2)$
4. $O(n^2)$
5. $O(n^4)$

References

[1] Sedgewick, R., & Wayne, K. (2014). Algorithms: part I (4th ed.). Pearson Education, Limited. `https://public.ebookcentral.proquest.com/choice/PublicFullRecord.aspx?p=7115411.`

[2] Pagh, R., & Rodler, F.F. (2004). Cuckoo hashing. *Journal of Algorithms*, *51*(2), 122-144. `https://doi.org/10.1016/j.jalgor.2003.12.002.`

[3] Broder, A. Z., & Mitzenmacher, M. (2003). Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, *1*(4), 485-509. `https://doi.org/10.1080/15427951.2004.10129096`

Appendix

```
import java.util.Arrays;
import java.util.LinkedList;

public class HashTable {

    private final int SIZE = 41;
    private final LinkedList[] table;

    private int collisions;

    public int getCollisions() {
        return this.collisions;
    }


    public LinkedList<Integer>[] getTable() {
        return table;
    }

    public HashTable() {
        table = new LinkedList[SIZE];
        for (int i = 0; i < SIZE; i++) {
            table[i] = new LinkedList();
        }
    }


    // first hashing function - k mod 16
    public int hash(Integer element) {

        return element % SIZE;

    }


    // second has function - Math.floor k / 16
    public int hash2(Integer element) {

        return element / SIZE;

    }



    public void insert(Integer element) {
        int hashed = hash(element);

        if ((table[hashed] != null) && (table[hashed].size() != 0)) collisions++;

        table[hashed].add(element);

    }
```

```
    public static void main(String[] args) throws InterruptedException {

        HashTable t = new HashTable();

        int elements = 40;
        for (int i = 0; i < elements; i++) {
            t.insert(i);

        }

        double averageKeysPerSlot = 0;
        int sizeForAvg = 0;
        for (int i = 0; i < t.getTable().length; i++) {

            if (t.getTable()[i].size() != 0) {
                averageKeysPerSlot += t.getTable()[i].size();
                sizeForAvg++;
            }

        }

        averageKeysPerSlot /= sizeForAvg;

        System.out.println("number of collisions: " + t.getCollisions());

        System.out.println(Arrays.toString(t.getTable()));

        System.out.println("----------------");

        System.out.println("Average num of keys per slot: " + averageKeysPerSlot);

        System.out.println("load factor: " + (double) elements/ t.SIZE);
    }
}
```