University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

Algorithms and Datastructures
Winter Term 2022
Grading Guidelines Exercise Sheet 4

Due: Wednesday, November 16th, 2pm

Exercise 1: Hashing with Open Addressing *(10 Points)*

We consider hash tables with open addressing and two different methods for collision resolution: *linear probing* and *double hashing*. Let m be the size of the hash table where m is prime. Let $h_1(x) := 53 \cdot x$ and $h_2(x) := 1 + (x \bmod (m-1))$. We define the following hash functions for collision resolution according to the lecture:

- linear probing: $h_\ell(x,i) := (h_1(x) + i) \mod m$.
- double hashing: $h_d(x,i) := (h_1(x) + i \cdot h_2(x)) \mod m$.

(a) Implement a hash table with operations `insert` and `find` using the mentioned strategies for collision resolution[1]. You may use the template `HashTable.py`. *(5 Points)*

(b) Create a hash table of size $m > 1000$ (m prime) and measure the average time for inserting k keys for $k \in \{\lfloor \frac{m \cdot i}{50} \rfloor \mid i = 1, \ldots, 49\}$ in four variations: Using linear probing / double hashing; inserting k random keys[2] / the set of keys $\{m \cdot i \mid i = 1, \ldots, k\}$. Create a plot showing the four different average runtimes. Discuss your results. *(5 Points)*

Sample Solution

(a) Cf. `HashTable.py` in the `public` repository.

(b) Cf. 1. To visualize all data in a single plot we used a logarithmic scale (upper plot) because of the large gaps between the different times for inserting. In the lower plot we visualized the variant with linear probing and deterministic input using a linear scale.

In the upper plot we can see that linear probing together with an input for which each key is mapped by h_1 to the same position is a worst case for this kind of collision resolution. The average time for inserting increases linearly because for the i-th key we need to iterate through i positions in the table to find a free position. This linear trend is also visualized in the lower plot.

With a randomized input, linear probing works well as long as the table is not getting too full. If it is too full (we fill it up to roughly 98%), the average time for inserting increases rapidly as it becomes likely to hit a large "cluster" of already filled table position.

For double hashing we see that the deterministic input which was bad for linear probing does not cause a problem. The average insertion time in this case does not differ significantly from the randomized input.

[1] You can assume that no more than m elements will be inserted to the hash table.

[2] Unique random values from $\{0, \ldots, z\}$ with $z \gg m$, e.g., with `random.sample(range(z+1), k)`.

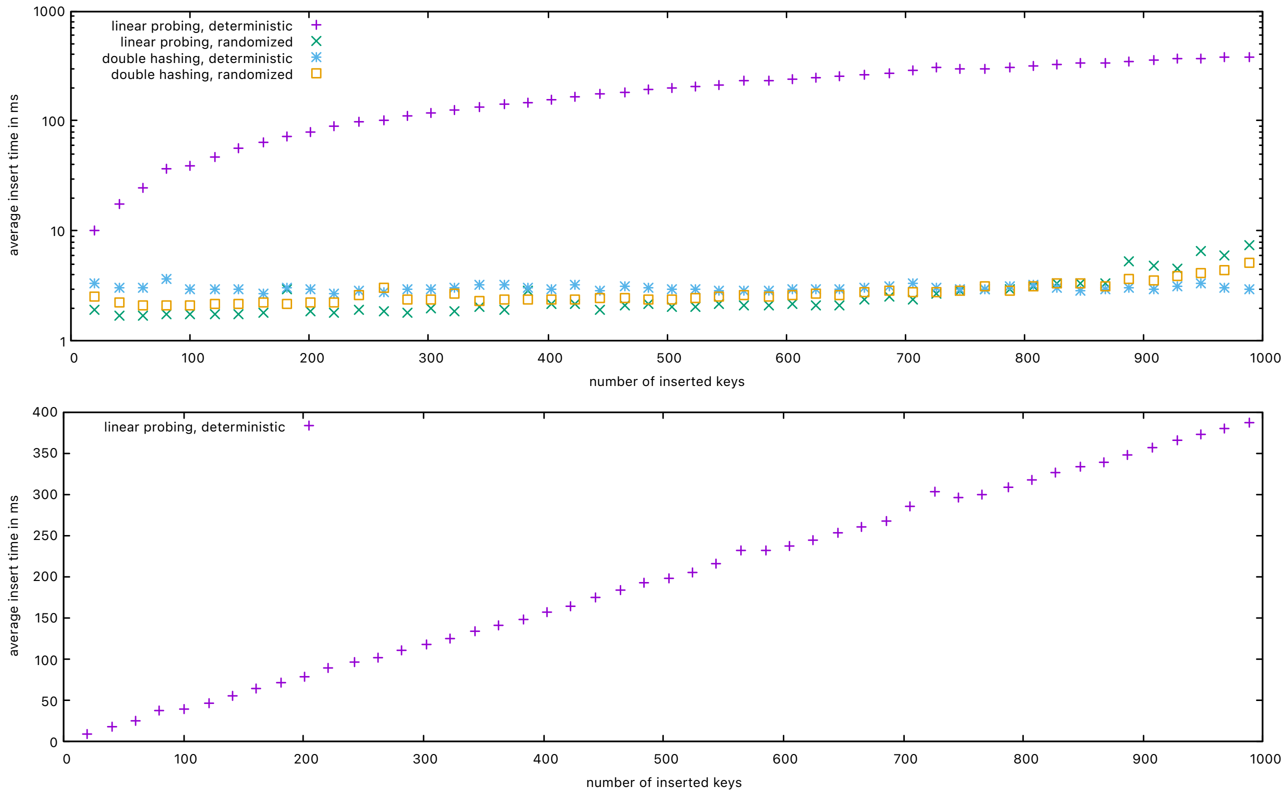

Abb. 1: Plots for exercise 1 b). Above: All variants on logarithmic scale. Below: Linear probing with deterministic input on linear scale.

Exercise 2: Application of Hashtables *(10 Points)*

Consider the following algorithm:

Algorithm 1 `algorithm` ▷ Input: Array A of length n with integer entries

```
for $i = 1$ to $n - 1$ do
    for $j = 0$ to $i - 1$ do
        for $k = 0$ to $n - 1$ do
            if $|A[i] - A[j]| = A[k]$ then
                return true
return false
```

(a) Describe what `algorithm` computes and analyse its asymptotical runtime. *(3 Points)*
Hint: The difference $|A[i] - A[j]|$ may become arbitrarily large.

(b) Describe a different algorithm $\mathcal{B}$ for this problem (i.e., $\mathcal{B}(A) =$ `algorithm`(A) for each input A) which uses hashing and takes time $\mathcal{O}(n^2)$ (with proof). *(3 Points)*

Hint: You may assume that inserting and finding keys in a hash table needs $\mathcal{O}(1)$ if $\alpha = \mathcal{O}(1)$ (α is the load of the table).

(c) Describe another algorithm for this problem without using hashing which takes time $\mathcal{O}(n^2 \log n)$ (with proof). *(4 Points)*

Sample Solution

(a) The algorithm checks if there are two entries in the array whose distance (absolute value of the difference) equals some entry in the array. If so, it returns "true", otherwise "false". In case it returns “false”, the algorithm runs completely through all three loops. It considers

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$

many pairs (i, j) and for each of this pair it checks n times the if-condition in line 4. Therefore, the runtime is $\mathcal{O}(n^3)$.

(b) We compute an array B of size $\mathcal{O}(n^2)$ which contains an entry $|A[i] - A[j]|$ for each pair (i, j) with $0 \leq j < i < n$. This takes time $\mathcal{O}(n^2)$. Afterwards we allocate a hash table of size $\mathcal{O}(n^2)$ (therefore $\alpha = O(1)$), choose a suitable hash function h and hash the values from B into the table H (this takes $\mathcal{O}(n^2)$ under ther assumption that one insert operation takes $\mathcal{O}(1)$). Finally, we test for each entry in A if it is contained in H, taking n times $\mathcal{O}(1)$. Hence the overall runtime is $\mathcal{O}(n^2)$.

(c) We sort A, taking $\mathcal{O}(n \log n)$. Afterwards we compute array B as in part (b), taking $\mathcal{O}(n^2)$. Now we test for each entry in B if it is in A using binary search. This takes n^2 times $\mathcal{O}(\log n)$. The overall runtime is dominated by the last step and equals $\mathcal{O}(n^2 \log n)$.