University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
P. Bamberger, P. Schneider

Algorithms and Data Structures
Summer Term 2019
Exercise Sheet 3

Exercise 1: Amortized Runtime of Dynamic Arrays

We want to implement a data structure D that stores elements consecutively in an array and supports an operation `append[`x`]` that writes x to the first non-empty entry in D (i.e., to $D[i]$ if $D[i-1]$ is the last non-empty entry in D). The array D has initial size 2 (i.e. it can hold 2 elements) but grows dynamically as we append more elements. Let n be the current number of elements in D ($n = 0$ when we initialize the data structure). Then `append[`x`]` does the following: If D.`size` $< n$ write $D[n] \leftarrow x$. Else create a new array D' of size $2n$ and copy all elements from D to D'.

Algorithm 1 `append[`x`]`

```
if D already contains n elements then
    create a new array D' of size 2n
    for i = 0 to n−1 do
        D'[i] ← D[i]
    D ← D';                                  ▷ "Rename" D' into D
D[n] ← x; n ← n + 1
```

Assume that creating a new array of size n takes n timesteps and writing an element into an array entry (e.g., $D[n] \leftarrow x$) takes 1 timestep. For simplicity you may assume everything else takes zero time. Starting with empty D, show that any series of `append[`x`]` operations has amortized running time $\mathcal{O}(1)$ per operation.

Exercise 2: Average Runtime[1]

The following algorithm obtains a number $x \in \{0, ..., n\}$. Additionally it obtains an array A of size $n+1$ that contains integers $\{0, ..., n\} \setminus \{x\}$ sorted in *ascending* order, whereas the last entry of A is empty. The algorithm inserts x into its position in A and moves the subsequent elements by one position.

Algorithm 2 INSERT($A[0..n]$, x)

```
i ← n
while A[i−1] > x do
    Swap A[i − 1] and A[i]
    i ← i − 1
A[i] ← x
```

Compute the average runtime for all possible inputs. To simplify things, assume that one swap operation takes one time unit, while all other operations have negligible runtime.

[1] This exercise was added for discussion in the exercise lesson

Exercise 3: Unsuitable Hash Functions

Let m be the size of a hashtable and let $n \gg m$ be the biggest possible key of any (key,value)-pair. A hash function $h : \{0, \ldots, n\} \to \{0, \ldots, m-1\}$ maps keys to table entries and should meet some criteria in order to be considered a suitable hash function.
The hash function should of course utilize the whole table, i.e., it should be a surjective function. Furthermore, it should be "chaotic", meaning that it should map similar keys to distinct table entries in order to avoid having lots of collisions in case many similar keys are inserted. A hash function must be deterministic. The following "hash functions" are unsuitable for various reasons. For each hash function quickly explain why this is the case.

(a) $h_1 : k \mapsto k$. [2]

(b) $h_2 : k \mapsto \lfloor \frac{k}{n} \cdot (m-1) \rfloor$.

(c) $h_3 : k \mapsto 2 \cdot \big(k \bmod \lfloor \frac{m}{2} \rfloor\big)$.

(d) $h_4 : k \mapsto \texttt{random}(m)$, $\big(\texttt{random}(m)$ is picked uniformly at random from $\{0, \ldots, m-1\}\big)$.

[2] The notation $h : k \mapsto h(k)$ means h maps the value k to the value $h(k)$.