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

Algorithms and Data Structures
Summer Term 2019
Sample Solution 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.

Sample Solution

The amortized runtime of a series of n operations can be computed as the total runtime of all operations divided by the number of operations. Let k be such that $2^k \leq n < 2^{k+1}$. Then we create a new array and copy all elements into it after $2, 2^2, 2^3, ..., 2^k$ steps respectively. The total cost for doing this in the 2^i-th ($i > 0$) append operation is $2^{i+1} + 2^i$. This is due to the fact that we create an new array of size 2^{i+1} in 2^{i+1} time steps, and then we copy 2^i elements into it in 2^i time steps. In total, all the necessary operations to double the arrays cost

$$T_{\text{doubling}}(n) = \sum_{i=1}^{k} 2^{i+1} + 2^i \leq \sum_{i=1}^{k} 2^{i+2} = \sum_{i=0}^{k-1} 2^{i+3} = 2^3 \cdot \sum_{i=0}^{k-1} 2^i = 2^3 \cdot (2^k - 1) \leq 2^{k+3}.$$

Additionally, for every append operation we write one element into the array which takes one time step, i.e. in total we have an additional $T_{\text{append}}(n) = n$ time steps. In total we have

$$T_{\text{total}}(n) = T_{\text{doubling}}(n) + T_{\text{append}}(n) \leq n + 2^{k+3} \leq 2^{k+1} + 2^{k+3} \leq 2^{k+4} \leq 2^4 \cdot 2^k \leq 2^4 \cdot n.$$

Divided by the number of operations we have obtain the amortized time $T_{\text{amort}}(n) = \frac{2^4 \cdot n}{n} = 16 \in \mathcal{O}(1)$.

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.

Sample Solution

The number of swaps that must be performed are equal to $n - x$. We have n+1 possibilities for choosing $x \in \{0, ..., n\}$. Therefore the average runtime (i.e., the average number of swap operations) is

$$T_{\text{avg}}(n) = \frac{\sum_{x=0}^{n}(n-x)}{n+1} = \frac{\sum_{x=0}^{n} x}{n+1} = \frac{n(n+1)}{2(n+1)} = \frac{n}{2}.$$

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 $h : \{0, \dots, n\} \to \{0, \dots, m-1\}$ 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) $h_3 : k \mapsto 2 \cdot \left(k \bmod \lfloor \frac{m}{2} \rfloor\right)$.

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

Sample Solution

(a) The given function is not even a valid hash-function, since it maps to values outside the table range.

(b) This hash function is not seperating similar valus well, that is if k_1, k_2 are close, then $h_2(k_1), h_2(k_2)$ are close as well. Since $n \gg m$ many successive keys are mapped to the same table entry.

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

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

(c) The function is not surjective, i.e. it does not use the whole address space (e.g. only even table entries are used, and the entry $m-1$ is never used). For $m=1$ the function is undefined (since mod 0 is undefined).

(d) A truly randomized function prohibits finding a value once it has been hashed.