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

Algorithms and Data Structures
Winter Term 2019/2020
Sample Solution Exercise Sheet 4

Remark: For this exercise, watch the relevant parts of the sixth and seventh video lecture.

Exercise 1: Hashing - Collision Resolution with Open Addressing

(a) Let $h(s,j) := h_1(s) - 2j \mod m$ and let $h_1(x) = x + 2 \mod m$. Insert the keys 51, 13, 21, 30, 23, 72 into the hash table of size $m = 7$ using linear probing for collision resolution (the table should show the final state).

0	1	2	3	4	5	6

(b) Let $h(s,j) := h_1(s) + j \cdot h_2(s) \mod m$ and let $h_1(x) = x \mod m$ and $h_2(x) = 1 + \big(x \mod (m-1)\big)$. Insert the keys 28, 59, 47, 13, 39, 69, 12 into the hash table of size $m = 11$ using the double hashing probing technique for collision resolution. The hash table below should show the final state.

0	1	2	3	4	5	6	7	8	9	10

(c) Repeat part (a) using the *"ordered hashing"* optimization from the lecture.

(d) Repeat part (b) using the *"Robin-Hood hashing"* optimization from the lecture.

Sample Solution

(a)

30	13	21	72	51	23	
0	1	2	3	4	5	6

(b)

	69	13	47	59	39	28	12			
0	1	2	3	4	5	6	7	8	9	10

(c)

30	13	21	72	23	51	
0	1	2	3	4	5	6

(d)

47 $j=1$	12 $j=0$	69 $j=1$	59 $j=1$	28 $j=1$	39 $j=1$	13 $j=1$				
0	1	2	3	4	5	6	7	8	9	10

Exercise 2: Amortized Analysis - Stack with Multipop

Consider the data structure "stack" in which elements can be stored in a *last in first out* manner. For a stack `S` we have the following operations:

- `S.push(`x`)` puts element x onto `S`.
- `S.pop()` deletes the topmost element of `S`. Assume `pop()` is only called if `S` is nonempty.
- `S.multipop(`k`)` removes the k top objects of `S`, popping the entire stack if `S` contains fewer than k objects.

Assume the costs of `S.push(`x`)` and `S.pop()` are 1 and the cost of `S.multipop(`k`)` is $\min(k, |\texttt{S}|)$ where $|\texttt{S}|$ is the current number of elements in `S`.

Use the bank account paradigm to show that all three operations have constant amortized cost. Assume that `S` is initially empty.

Sample Solution

Define the costs of the operations as follows:

operation	amortized cost	actual cost
`S.push(`x`)`	2	1
`S.pop()`	0	1
`S.multipop(`k`)`	0	$\min(k, \lvert\texttt{S}\rvert)$

The difference between the amortized cost minus the actual cost can be seen as the number of coins that are "paid" into or subtracted from the bank account whenever the corresponding operation occurs. We have to show that for a sequence of n operations (under the assumption that the stack is initially empty and that we never call `S.pop()` on the empty stack), we always have enough coins on the bank account to "pay" said difference between amortized and actual cost in case the difference is negative (which is the case for the latter two operations).
For the data structure above this is fairly obvious, as we see from the following, simple argument. For each `S.push(`x`)` operation we increase the bank account by one. Since we can remove at most as many elements as we pushed before (initially we assume an empty stack) and since each removed element is associated with a cost of 1 (no matter if we remove it with `S.pop()` or in a batch with `S.multipop(`k`)`) we have always enough coins on the bank account to pay for the removal. That is, if c_i the actual cost and a_i the amortized cost of operation $i \leq n$, then

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i.$$

Exercise 3: Amortized Analysis - a Hierarchy of Arrays

Consider the following data structure. We define arrays A_i (for $i = 0, 1, 2, \ldots$), where A_i has size 2^i and stores integer keys in a sorted manner (ascending). During the runtime we ensure that each Array is either completely full, or completely empty.
We informally describe an operation `insert(`k`)`. It first tries to insert the key k into A_0. If A_0 is empty, we insert k into A_0 and are done. If A_0 happens to be already full (i.e. it contains one element), A_0 is *merged* with k to form a new sorted array B_1 of size 2. If A_1 is empty, B_1 becomes the new Array A_1 and we are done. Else B_1 is merged with A_1 into a sorted Array B_2 of size 4 and the same procedure is repeated with $A_2, A_3, \ldots$ until we find an Array A_i that is empty.

(a) Describe a subprocedure `merge`(A, B) (as pseudo code or as informal algorithm description) that merges the contents of two sorted Arrays A, B of size m into a new, sorted array of size $2m$ in $\mathcal{O}(m)$ runtime. Explain why your algorithm has the runtime $\mathcal{O}(m)$.

(b) Show that any series of n `insert`-operations has an *amortized* runtime of at most $\mathcal{O}(\log n)$.

Sample Solution

(a) Consider the following pseudocode.

Algorithm 1 `merge`$(A[0..m-1], B[0..m-1])$

```
C ← allocate array of size 2m
i ← 0; j ← 0; k ← 0
while k < 2m do
    if j = m or A[i] ≤ B[j] then
        C[k] ← A[i]; i ← i + 1
    else
        C[k] ← B[j]; j ← j + 1
    k ← k + 1
return C
```

The loop has $2m \in \mathcal{O}(m)$ cycles since k is increased in each cycle, and in each cycle we do only constant time operations.

(b) We want to compute the sum of runtime of all n `insert` operations. Let T_{total} be the total runtime for inserting n elements. As the `merge` operation is the costliest operation during each `insert`, the overall runtime T_{total} equals the runtime for the overall number of `merge` operations occurring during *all* inserts. We make a few observations about these `merge` operations.

Observation 1. An Array A_i of size $2^i > n$ is never used since it would not be completely filled. Hence only arrays A_i with $2^i \leq n$ (i.e. $i \leq \lfloor \log_2(n) \rfloor$) can be non-empty at some point.

Observation 2. Elements are only moved from smaller to larger arrays, when we merge two arrays into the next bigger array. Hence, each element is inserted at most once into each array A_i.

Observation 3. We always insert 2^i elements at once into A_i and we have at most n elements in our data structure. Since each element is inserted at most once on each level i (Observation 2), we can conduct the associated merge operation of arrays A_{i-1}, B_{i-1} at most $n/2^i$ times.

Observation 4. The cost of merging arrays A_{i-1}, B_{i-1} is $\mathcal{O}(2^i)$ (c.f., part (a)).

Let T_i be the total cost of *all* `merge` operations done on level i. We calculate T_{total} as follows

$$T_{\text{total}} = \sum_{i=0}^{\text{“}\infty\text{”}} T_i \overset{\text{Obs. 1}}{=} \sum_{i=0}^{\lfloor \log_2 n \rfloor} T_i \overset{\text{Obs. 3+4}}{=} \sum_{i=0}^{\lfloor \log_2 n \rfloor} \tfrac{n}{2^i} \cdot \mathcal{O}(2^i) = \sum_{i=0}^{\lfloor \log_2 n \rfloor} \mathcal{O}(n) = \mathcal{O}(n \log n)$$

Since we have n operations, the amortized cost of one `insert` operation is

$$T_{\text{insert}} = \frac{T_{\text{total}}}{n} = \mathcal{O}(\log n).$$