University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
S. Faour, M. Fuchs, A. Malyusz

Algorithm Theory
Sample Solution Exercise Sheet 5

Due: Friday, 24th of November 2023, 10:00 am

Exercise 1: Amortized Analysis *(4+4+4 Points)*

Your plan to implement a `Stack` with the classical operations `push`, `pop` and `peek`. As underlying data structure you use a dynamic array that will grow its size whenever 'many' elements are stored and on the other hand also shrinks its size when only a view elements remain in the array. In the following let n_i be the number of elements stored in the array and let s_i be the size of the array after the i-th operation.

- Before you **push** a new element x to the array, you check if $n_{i-1} + 1 < 80\% \cdot s_{i-1}$. If this is the case then you simply add x. We say for simplicity, that this can be done in 1 time unit. If on the other hand $n_{i-1} + 1 \geq 80\% \cdot s_{i-1}$, you set up a new (empty) array of size $s_i := 2s_{i-1}$ and copy all elements (and x) into the new one. We assume this can be done in s_{i-1} time units[1].
- To **pop** an element from the array, you first check if $n_{i-1} - 1 > 20\% \cdot s_{i-1}$. If this is the case then pop x within 1 time unit. If the table size is small, say $s_{i-1} \leq 8$, you also just pop x. But, if $n_{i-1} - 1 \leq 20\% \cdot s_{i-1}$ and $s_{i-1} > 8$, create a new (empty) array of size $s_i := s_{i-1}/2$ and copy all values except x into this new array. By assumption, this step takes s_i time units.
- The **peek** operation returns the last inserted element in 1 time unit. Note that state of the array does not change, i.e., $n_{i-1} = n_i$ and $s_{i-1} = s_i$.

Initially, the array is of size $s_0 = 8$. Assume that this initial step can also be done in 1 time unit. Note that by this initial size and the definition of the pop method we have $s_i \geq 8$ for all $i \geq 0$. Also note that after every operation that resized the array at least one element can be pushed or popped until a further resize is required.

a) Let i be a push operation that resized the array. Show that the following holds.

$$0.4 \cdot s_i \leq n_i < 0.55 \cdot s_i$$

Further, show that if i is a pop operation that resized the array, the following holds.

$$0.25 \cdot s_i < n_i \leq 0.4 \cdot s_i$$

b) Use the `Accounting Method` from the lecture to show that the **amortized running times** of push, pop and peek are $O(1)$, i.e., state by how much you additionally charge these three operation and show that the costs you spare on 'the bank' are enough to pay for the costly operations.
Hint: Use the previous subtask, even if you didn't manage to show them.

c) Show the same statement as in the previous task, but use the `Potential Function Method` this time, i.e., find a potential function $\phi(n_i, s_i)$ and show that this function is sufficient to achieve constant amortized time for the supported operations.
Hint: There is not just one but infinitely many potential functions that work here. However, you may want to use a function of the form $c_0 \cdot |n_i - c_1 \cdot s_i|$ for some properly chosen constants $c_0 > 0$ and $c_1 > 0$.

[1] For a simpler calculation we use normalized time units, such that all the operations that would take $O(1)$ time will take at most 1 time unit and operations that would take $O(s_{i-1})$ time will take at most s_i time units.

Sample Solution

a) **Push:** It is clear that for the previous state $n_{i-1} < 0.8s_{i-1}$ is true (otherwise there would have been a resize before) and by definition also $n_{i-1}+1 \geq 0.8s_i$ holds. Since $n_i = n_{i-1}+1$ and $s_i = 2s_{i-1}$ we directly get $0.4 \cdot s_i \leq n_i < 0.4 \cdot s_i + 1 \leq 0.525 \cdot s_i$ (because $s_i \geq 8$). This implies the statement.
Pop: For similar reasons as before, we have $0.2 \cdot s_{i-1} < n_{i-1} \leq 0.2 \cdot s_{i-1} + 1$. Substituting $n_i = n_{i-1} - 1$ and $s_i = s_{i-1}/2$ we get $0.4 \cdot s_i < n_i + 1 \leq 0.4 \cdot s_i + 1$. By subtracting all sides by 1 and use that $1 \leq s_i/8$ we get $0.275 \cdot s_i < n_i \leq 0.4 \cdot s_i$. This implies the statement.

b) We charge all 3 operations by 25 'dollars'. Every push, pop or peek operation costs one actual dollar (not counting resizing) and puts the remaining 24 in the bank to pay for resizing. Now, let us estimate how much money is **at least** in the bank before the next resizing takes place. Observe that we can ignore peek operations in the following, since they just increase our bank account (by 24 per operation) and do not change the state of the array. Let us for now assume that the last operation (say operation i) did a resizing and currently there is no money at our account. If this last resize operation was 'push', then we have $0.4 \cdot s_i \leq n_i < 0.55 \cdot s_i$. Thus, the next costly operation can not happen before $(0.8 - 0.55)s_i = 0.25s_i$ push or $(0.4 - 0.2) \cdot s_i = 0.2s_i$ pop operations. Before we proceed, let us see how it looks if operation i was a pop operation. By the statement of the previous task we have the next costly operation not before $(0.8 - 0.4) \cdot s_i = 0.4s_i$ push or $(0.25 - 0.2) \cdot s_i = 0.05 \cdot s_i$ pop operations. By this analysis, the *worst-case* (i.e., the shortest chain of operation until the next resize) is a costly pop operation that is followed by more than $0.05 \cdot s_i$ additional deletings. Since we charge both operation with the same amortized cost, we clearly have more 'money' on our account in the other cases. Also observe that if we alternate between pushing and deleting over and over, the hash table is never resized, so we save up a lot of money in the bank. For that, assume operation i is a pop operation and from here on at least $0.05 \cdot s_i$ further pop operations follow until the next resize comes. The money on the bank after these many operations is at least $(25 - 1) \cdot 0.05 \cdot s_i = 1.2 \cdot s_i$. Because the costly operation costs $1 \cdot s_i$, we can afford it and thus, the bank account never drops below zero. Therefore, the amortized cost of pop is $O(1)$. This, by above's analysis, also implies amortized costs of $O(1)$ for push (as well as for peek).

c) We define our potential function by $\phi(n_i, s_i) := c_0 \cdot |n_i - c_1 \cdot s_i|$ and start by guessing some constant $c_1 \in [0, 1]$. For some intuition: We want our potential to be large before a resizing operation and small (close to zero) after a resizing operation. To make sure that the potential before resizing is not 0, we have to choose $c_1 \neq 0.8$ in the push case and $c_1 \neq 0.2$ in the pop case. So let us 'guess' $c_1 = 0.4$ and show later that it works. Now we go through all operations and proof that by choosing a large enough $c_0 > 0$, all amortized costs are in $O(1)$. Note that we are going into 5 cases now, that we call `peek`, `cheap push` (push without resizing), `costly push` (includes resizing), `cheap pop` (deleting without resizing) and `costly pop` (includes resizing). Like in the lecture, we notate the actual cost of operation i by t_i and its amortized costs by $a_i := t_i + \phi(n_i, s_i) - \phi(n_{i-1}, s_{i-1})$.

Peek Here we have $s_i = s_{i-1}$ and $n_i = n_{i-1}$ and thus,

$$\begin{aligned} a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\ &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 0.4s_i| \\ &= 1 \end{aligned}$$

Cheap Push Here we have $n_i = 1 + n_{i-1}$ and $s_i = s_{i-1}$

$$\begin{aligned} a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\ &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 1 - 0.4s_i| \\ &\leq 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot (|n_i - 0.4s_i| - 1) \\ &\leq 1 + c_0 \end{aligned}$$

Note that this implies that $a_i \in O(1)$ if c_0 is a fixed constant.

Costly Push Here we have $t_i = s_{i-1}$, $s_i = 2s_{i-1}$, $n_i = n_{i-1} + 1$ and by the first subtask $0.55s_i > n_i \geq 0.4s_i$

$$\begin{aligned}
a_i &= s_{i-1} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\
&= \frac{s_i}{2} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot (|n_i - 0.2s_i| - 1) \\
&\leq \frac{s_i}{2} + c_0 \cdot (n_i - 0.4s_i) - c_0 \cdot (n_i - 0.2s_i) + c_0 \\
&\leq \frac{s_i}{2} - \frac{1}{5} \cdot c_0 \cdot s_i + c_0 \\
&\leq c_0
\end{aligned}$$

That the last step only follows if we choose $c_0 \geq \frac{5}{2}$.

Cheap Pop Here we have $n_i = -1 + n_{i-1}$ and $s_i = s_{i-1}$

$$\begin{aligned}
a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\
&= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i + 1 - 0.4s_i| \\
&\leq 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 0.4s_i| + c_0 \\
&\leq 1 + c_0
\end{aligned}$$

Costly Pop Here we have $t_i = s_{i-1}$, $s_i = s_{i-1}/2$, $n_i = n_{i-1} - 1$ and by the first subtask $n_i > 0.25s_i$ and $n_i \leq 0.4s_i$.

$$\begin{aligned}
a_i &= s_{i-1} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\
&= 2s_i + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i + 1 - 0.8s_i| \\
&\leq 2s_i + c_0 \cdot (0.4s_i - n_i) - c_0 \cdot (0.8s_i - n_i) + c_0 \\
&= 2s_i - \frac{2}{5} \cdot c_0 \cdot s_i + c_0 \\
&\leq c_0
\end{aligned}$$

That the last step only follows if we choose $c_0 \geq 5$.

Final Statement It is clear by the previous calculations that if we choose $c_0 := 5$ the amortized costs for all 3 operations are at most $a_i \leq 1 + c_0 = 6$ and therefore constant. The potential function used is

$$\phi(n_i, s_i) := 5 \cdot |n_i - \frac{2}{5} \cdot s_i| = |2 \cdot s_i - 5 \cdot n_i| \geq 0$$

Remark: We have that $\phi(n_0, s_0) = 5 \cdot |0 - \frac{2}{5} \cdot 8| = 16$ and hence $\sum_i t_i \leq 16 + \sum_i a_i$. This does not completely match with the definition of amortized costs, however we can fix this problem by choosing the potential function $\phi'(n_i, s_i) := 5 \cdot |n_i - \frac{2}{5} \cdot s_i - \frac{16}{5}|$. Here we have $\phi'(n_0, s_0) = 0$. Further we have that for all i, $|\phi'(n_i, s_i) - \phi(n_i, s_i)| \leq 16$ holds, and thus we can simply adjust the 5 cases and show that for each operation it follows $a_i \leq t_i + \phi(n_i, s_i) - \phi(n_{i-1}, s_{i-1}) + 2 \cdot 16 \leq 6 + 32 = 38$.

Exercise 2: Union-Find *(2+2+4 Points)*

In the lecture we have seen two heuristics (i.e., the **union-by-size** and the **union-by-rank** heuristic) to implement the **union-find** data structure. In this exercise we will focus on the **union-by-rank** heuristic only! Note that the rank is basically the height of the underlying tree. This is not true if we use *path compression* as the height of the tree might change; but the rank is still an upper bound on the actual height of the tree. To solve the following tasks consider the **union-find** data structure implemented by disjoint forest using union-by-rank heuristic and path compression.

(a) Give the pseudocode for `union(x, y)`.
Remark: Use $x.parent$ to access the parent of some node x and use $x.rank$ to get its rank. The `find(x)` operation is implemented as stated in the lecture using path compression.

(b) Show that the height of each tree (in the disjoint forest) is at most $O(\log n)$ where n is the number of nodes.

(c) Show that the above's bound is tight, i.e., give an example execution (of `makeSet`'s and `union`'s) that creates a tree of height $\Theta(\log n)$. Proof your statement!

Sample Solution

(a) The pseudocode is given below.

Algorithm 1 `union`(x, y)

```
a := find(x)
b := find(y)
if a.rank > b.rank then
    b.parent := a
    return a                    ▷ returning the root of the new set (optional)
else
    a.parent = b
    if a.rank = b.rank then
        b.rank = b.rank + 1
    return b                    ▷ returning the root of the new set (optional)
```

(b) Since the rank is an upper bound on the height of a tree, we will show that the maximum rank of a tree is $O(\log n)$. We will show by induction over the rank of the tree that for any tree T with $T.rank = r$, T contains at least $|T| \geq 2^r$ many nodes. We start our induction at $r = 0$. Surely, only trees with exactly one node can have a rank of 0, what fulfills the condition.

Induction Hypothesis: For a fixed r, any tree T_r of rank r contains at least 2^r nodes.

Induction Step: Let T_{r+1} be a tree with rank $r+1$. Since the rank of a tree increases only then when it is merged with another tree of equal rank, we can say w.l.o.g that T_{r+1} was created by the union of trees T_r and T_r', both with rank r. Due to our hypothesis, we know $|T_r| \geq 2^r$ and $|T_r'| \geq 2^r$. Since T_{r+1} contains at least the nodes of T_r and T_r' we have

$$|T_{r+1}| \geq |T_r| + |T_r'| \geq 2^r + 2^r = 2^{r+1}$$

which ends the inductive proof.

We can conclude that for any tree T_r: $n \geq |T_r| \geq 2^r \Rightarrow height(T_r) \leq r \leq \log_2 n$.

(c) For simplicity, assume we have $n = 2^k$ nodes that we add to our union-find data structure using `makeSet`. We now have n trees of rank 0. Next, we `union` all pairs of trees s.t. we get $n/2$ trees of rank 1. Again, we `union` over all pairs of trees and get $n/4$ trees of rank 2. Continuing like this, we will finally get a single tree with rank k. Note that even with *path compression*, the rank of these trees is equal to its height, since we can always use the roots of each tree for our `union`-method from (a) and therefore the find operation will not rearrange the pointers of the children. Hereby our execution leads to a tree with $height = k = \log_2 n$.

Note that if n is not a power of 2, we can still use the same construction and finally get 2 trees where the larger one contains more than $n/2$ of the nodes. This tree's height is $\log_2 n - 1 = \Theta(\log n)$.