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

Algorithm Theory
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.

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!