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

Algorithm Theory
Exercise Sheet 5

Due: Wednesday, 30th of November 2022, 11:59 pm

Exercise 1: Potential Function Method *(12 Points)*

You plan to implement a hash table. Since you do not know how many keys will be inserted (or deleted) into that hash table, you implement the hash table such that it's size dynamically grows/shrinks depending on the number of keys that are currently stored. To archive that property, you implement an **insert** and a **delete** function that work as follows. Let n be the number of keys in the hash table and let s be the current table size.

- Before you **insert** a new key x to the table, you check if $n < \frac{4}{5} \cdot s$. If this is the case then you simply add x. We say for simplicity this can be done in 1 time unit. If on the other hand $n \geq \frac{4}{5} \cdot s$, you set up a new hash table of size $2s$ and paste all keys (including x) into the new table. We assume this can be done in s time units[1].
- To **delete** an already stored key x from the hash table, you first check if $n > \frac{1}{5} \cdot s$. If this is the case then delete x within 1 time unit. If otherwise $n \leq \frac{1}{5} \cdot s$ and $s > 10$, create a new hash table of size $s/2$ and copy all values except x to the new table. By assumption, this step takes s time units.

Initially, the hash table is empty. When the fist key x is added, you build an initial table of fixed size, say $s_0 := 10$, and insert x. Assume that this initial step can also be done in 1 time unit. Note that by this initial size and the definition of the delete method we have $s \geq 10$ at any point.

Your task is to show that the **amortized running times** of insert and delete are $O(1)$. To do that, use the *Potential Function Method* from the lecture i.e., find a potential function $\phi(n, s)$ and show that this function is sufficient to achieve constant amortized time for any insert and delete operation.

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. In the following tasks assume a disjoint forest implementation 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.

[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 $O(s)$ time will take at most s time units.

(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!