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

Algorithms Theory
Sample Solution Exercise Sheet 5

Due: Tuesday, 8th of December 2020, 4 pm

Exercise 1: Amortized Analysis *(10 Points)*

Consider a binary min-heap data structure that supports the two operations `insert` and `delete-min`. The heap is initially empty and we assume that its number of elements never exceeds n.

(a) Use the *accounting* method to show that we can consider the amortized cost of `insert` to be $O(\log n)$ and the amortized cost of `delete-min` to be $O(1)$. *(3 Points)*

(b) Show the statement from part (a), this time using the *potential function* method. *(5 Points)*

(c) We would like to amortize the costs differently such that the amortized cost of `insert` is $O(1)$ and the amortized cost of `delete-min` is $O(\log n)$. Either define a feasible *potential function* that yields these amortized costs or argue why this is not possible. *(2 Points)*

Sample Solution

(a) Let $f(n) \in O(\log n)$ be the maximum (actual) cost of `insert` and `delete-min`. We define the amortized cost of `insert` as $2f(n)$ and the amortized cost of `delete-min` as 0. In any sequence of operations on an initially empty heap, there are more `insert` than `delete-min` operations (assuming one can not call `delete-min` on an empty heap) and so our bank account is always non-negative or, in other words, the sum of the amortized costs is at least the sum of the actual costs.

If we also allow to call `delete-min` on an empty heap with actual cost $c = O(1)$, we can set the amortized cost of `delete-min` to c. Then the charged costs for the `insert` operations pay for the `delete-min` operations that are called on a non-empty heap.

(b) Let $f(n) \in O(\log n)$ be the maximum (actual) cost of `insert` and `delete-min`. We define a potential function $\phi = f(n) \cdot$ #elements in the heap. For any sequence of operations starting on an empty heap we have $\phi_0 = 0$ and $\phi_i \geq 0$ and thus ϕ is a valid potential function. The amortized cost of `insert` (actual cost plus change of potential) is $\leq 2f(n)$ and the amortized cost of `delete-min` is ≤ 0.

(c) A sequence of n `insert` operation has actual cost $\sum_{i=1}^{n} \log i = \omega(n)$, so there is no way to obtain constant amortized cost for `insert`.

Exercise 2: Union Find - Linked List Implementation *(8 Points)*

In the lecture, we have seen a linked list implementation where each linked list has a pointer to the first *and* last element. Describe an alternative implementation that uses only *one* of these pointers. Your scheme should still allow for the union-by-size heuristic and should not increase the asymptotic running time of the operations.

Sample Solution

We can omit the pointer to the first element. If we take the first element as representative, we can access the first element from any other element in O(1), so a pointer to the last element is sufficient.

Alternatively, we can omit the pointer to the last element. What we have to do is to adjust the `union` operation, i.e., the merging of two lists. Given two linked lists L and S where $|L| \geq |S|$, we insert S between the first and the second element of L. More detailed, we redirect the pointer of the first element of L to the first element of S, then we go through S and reset the representative pointer of each element to the representative (first element) of L. When reaching the last element of S, we set its pointer to the second element of L. The running time is linear in the length of S as in the case with two pointers.

Remark: Be careful to distinguish between the different pointers. There are pointers associated with the lists (pointing to the first element) and for each list element (except the last) a "next element pointer" and a "representative pointer".

Exercise 3: Union Find - Disjoint-Set Forests *(8 Points)*

(a) Give a sequence of m `make-set`, `union`, and `find` operations, n of which are `make-set` operations, that takes $\Omega(m \log n)$ time when we use union by rank only. *(3 Points)*

(b) Suppose that we wish to add the operation `print-set`, which is given a node x and prints all the members of x's set, in any order. Show how to add this feature to the disjoint-set forest implementation such that `print-set` takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that we can print each member of the set in $O(1)$ time. *(5 Points)*

Sample Solution

(a) Let $m = 3n$. We do n `make-set` operations `make-set`$(x_0), \ldots,$ `make-set`(x_{n-1}) and then the following `union` operations (for simplicity we assume that n is a power of 2):

---

for $i = 1$ to $\log n$ **do**
  $k = 2^i$
  for $j = 1$ to $\frac{n}{k}$ **do**
    `union`$(x_{(j-1)\cdot k}, x_{j\cdot k-1})$

We obtain one tree with depth $\log n$ and x_{n-1} as a leaf. Now we call n times `find`(x_{n-1}) which costs $\Omega(n \log n) = \Omega(m \log n)$.

(b) Additionally to the parent pointer, we store at each element a doubly linked list of its children. Then we implement `print-set`(x) by first finding the root and then visiting (and printing) all nodes of the tree using BFS.

We must adapt the `make-set`, `union`, and `find` operations such that the childlist of each node is maintained and the asymptotic running time does not change. When calling `make-set`(x), we allocate an empty list for x's children, so the costs are still $O(1)$. The `find` operation is adapted in the following way

Algorithm 1 `find`(a)

```
1: if a ≠ a.parent then
2:     old_parent = a.parent
3:     delete a from the childlist of old_parent
4:     r = find(old_parent)
5:     a.parent = r
6:     Add a to r's childlist
7: return a.parent
```

That is, when we reset the parent pointer of a node a to the root r when doing path compression, we add a to r's childlist and delete it from its old parent's childlist. This yields constant additional cost in each step, so the cost of `find` is multiplied with a constant factor (note that storing each node's children in a *doubly* linked list makes it possible to efficiently delete a node from it). When attaching a tree with root r to a tree with root r' in a union operation, we add r to the childlist of r', leading to constant additional cost for union (when ignoring the cost for the `find` subroutine).

Alternatively, instead of storing the children of each node, we can maintain a linked list of the elements additional to the tree structure, with the root as the first element of the list. For efficient concatenation, we need an additional pointer from the first element of the list to the last one. Then, when attaching T_y (the tree containing y) to T_x (the tree containing x) in a `union`(x, y) call, we attach the list of nodes of T_y to the list of nodes of T_x with $O(1)$ overhead (we have to find the roots r_x and r_y of T_x and T_y anyway, and then we go from r_x to the last element in the list and set a pointer from this element to r_y). A `find` operation (with path compression) only changes the tree structure and therefore needs no adaptation. When calling `print-set`(x) we go to the root and then through the list of elements.