Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, O. Saukh, A. R. Molla December 2, 2015

Algorithm Theory, Winter Term 2015/16
Problem Set 6 - Sample Solution

Exercise 1: Amortized Analysis (4 points)

We are given a data structure $\mathcal{D}$, which supports the operations `put` and `flush`. The operation `put` stores a data item in $\mathcal{D}$ and has a running time of 1. Further, if $\mathcal{D}$ contains $k \geq 0$ items, the operation `flush` deletes $\lceil k/2 \rceil$ of the k data items stored in $\mathcal{D}$ and its running time is equal to k.
Prove that both operations have constant amortized running time by using the potential function method.

Solution

Goal:
$\mathcal{O}(1)$ amortized time per operation.

Intuition:
Flushing every item costs at most 2. Hence, a potential function should be increased by at least 2 whenever we `put` an item. Therefore, we are guaranteed to have enough potential to do a `flush` operation.

Definition of potential function:
A potential function is a function mapping a *possible configuration of the data structure* to a non negative real number. It is important that the potential function can be computed from the status of the data structure. In particular the information about previous operations that have been performed is not necessary to determine its value.
Define $\Phi = 2N$, where N is the current number of elements in the data structure $\mathcal{D}$.

Correctness of potential function:
The above potential function is never less than 0 since the number of elements in $\mathcal{D}$ can not be negative.

Amortized cost for i-th operation:
For the analysis, fix an arbitrary sequence of operations. Then let N_{i-1} denote the number of elements in $\mathcal{D}$ before the i-th operation and N_i the number of elements after the i-th operation [1]. In the following a_i and t_i are amortized and actual costs, respectively, for operation i. For the i-th operation we consider the cases that it is a `put` or a `flush` operation separately:

If the i-th operation is `put`, then:

$N_i = N_{i-1} + 1$.
$t_i = 1$ (the actual cost of operation `put`).
$a_i = t_i + \Phi_i - \Phi_{i-1} = 1 + 2N_{i-1} + 2 - 2N_{i-1} = 3 \in \mathcal{O}(1)$.

[1] N_i depends on the type of operation that is performed in step i. Thus N_i differs in the two considered cases

If the i-th operation is `flush`, then:

$N_i = N_{i-1} - \lceil N_{i-1}/2 \rceil$.
$t_i = N_{i-1}$ (the actual cost of operation `flush`) as given in the exercise.
$a_i = t_i + \Phi_i - \Phi_{i-1} = N_{i-1} + (2N_{i-1} - 2\lceil N_{i-1}/2 \rceil) - 2N_{i-1} = N_{i-1} - 2\lceil N_{i-1}/2 \rceil \leq 0 \in \mathcal{O}(1)$.

Hence, the amortized costs for both operations `put` and `flush` are constant.

Exercise 2: Union-Find (4+4 points)

(a) In the lecture the union-by-size heuristic was introduced to guarantee shallow trees when implementing a Union-Find data structure. Another heuristic that can be used for `union(x,y)` is the union-by-rank heuristic. For the heuristic, the rank of a tree is defined as follows:

- The rank $r(T)$ of a tree T consisting of only one node is 0.
- When joining trees T_1 and T_2 by attaching the root of tree T_2 as a new child of the root of tree T_1, the rank of the new combined tree T is defined as $r(T) := \max\{r(T_1), r(T_2) + 1\}$.

When applying the union-by-rank heuristic, whenever combining two trees into one tree (as the result of a union operation), we attach the tree of smaller rank to the tree of larger rank (if both trees have the same rank, it does not matter which tree is attached to the other tree). Provide pseudo-code for the `union(x,y)` operation when using the union-by-rank heuristic.

Show that when implementing a Union-Find data structure by using disjoint-set forests with the union-by-rank heuristic, the height of each tree is at most $O(\log n)$.

(b) Demonstrate that the above analysis is tight by giving an example execution (of merging n elements in that data structure) that creates a tree of height $\Theta(\log n)$. Can you even get a tree of height $\lfloor \log_2 n \rfloor$?

Solution

a) $p[x]$ denotes the parent of node x, and $h[x]$ denotes the rank of node x. The algorithm returns the representative of the set `union(x,y)`.

Algorithm 1: Union(x,y)

```
r := Find(x);
q := Find(y);
if r == q then
    return r;
if h[q] > h[r] then
    p[r] = q;
    return q;
else
    p[q] = r;
    if h[q] = h[r] then
        h[r] := h[r] + 1;
    return r;
```

Given a tree T in the union-find data structure with rank $h = h(T)$ one needs to show that $|T| \geq 2^h$.[2] We prove this result by induction on the rank of the tree:

[2] If we do not use path compression the rank equals the height. As the height only becomes smaller with the help of path compression the result still holds for union-find data structure with path compression.

Induction Start: Trees of rank 0 do have 1 node.
Induction Hypothesis: Let h be arbitrary but fixed such that $|T_k| \geq 2^k$ holds for all trees T_k with rank $k \leq h$.
Induction Step: Now let T be any tree with rank $h+1$. Then T was created by the union of two trees T_1 and T_2 both with rank h. (This point is essential: The union procedure is the only one which changes the rank. The rank can only grow when in the union of two trees with equal rank. The rank increases at most by one.).

$$|T| = |T_1| + |T_2| \overset{I.H.}{\geq} 2^h + 2^h = 2^{h+1}.$$

The result follows by induction.

b) Let us first define a binomial tree. A binomial tree B_k, for any $k \geq 0$, is defined as follows:
B_0 is a single node. For any $k > 0$, B_k is a tree with a single node (as the root of the tree) which has k subtrees, $B_0, B_1, \ldots, B_{k-1}$.

A combination of Binomial trees will lead to a configuration of the union-find data structure of height $\log_2(n)$, answering both questions in the affirmative:
Start with $n = 2^k$ trees B_0, combine them to $n/2$ trees B_1. Grouping them in pairs and combining them again will give $n/4$ trees of type B_2. Continuing like that will result in a single B_k, which, as we know from the lecture, has height $k = \log_2 n$. This gives possitive answer to both questions.
Let n be a power of 2. Then a sequence of operations is given by:

Algorithm 2: Results in a Union-Find Data Structure Configuration with height $\log_2(n)$

```
/* Create the trees B_0                                          */
for i=0 to n-1 do
    makeSet(i);
/* Merge the trees.                                              */
for i = 1 to log_2(n) do
    for j = 0 to n/2^i - 1 do
        union(j · 2^i, j · 2^i + 2^(i-1));
```

Notice that the `find` operation (**in this example execution!**) is always performed on the representative of a set. Thus the path compression is never used.