Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, O. Saukh, A. R. Molla November 11, 2015

Algorithm Theory, Winter Term 2015/16
Problem Set 5 - Sample Solution

Exercise 1: Amortized Analysis (2+4 points))

(a) Consider an extension of the augmented stack data structure from the lecture: In addition to offering a *multipop*(k) operation, assume that the extended augmented stack also offers a *multipush*(k, L) operation which takes a parameter $k \geq 1$ and a list L of k elements and it pushes each of these elements on the stack. Assume that both operations *multipop*(k) and *multipush*(k, L) require time $\Theta(k)$ to complete. In the lecture, we proved that for the simple augmented stack, the amortized cost of all operations is $O(1)$. Does this $O(1)$ amortized cost bound for all stack operations continue to hold for the extended version of the augmented stack? Explain your answer!

(b) Show how to implement a queue with two ordinary stacks so that the amortized cost of each *enqueue* and *dequeue* operations is $O(1)$. Explain your amortized analysis.

Solution

(a) Based on any standard implementation of stack operation we can assume that the actual cost for any *multipush*(k, L) or *multipop*(k) operation is $\Theta(k)$ and for any *push*() or *pop*() operation is $O\Theta(1)$.

For the sake of contradiction let us assume that the amortized cost of any of the four above operations in any sequence of these operations is constant and is bounded by some positive integer c. Let the actual cost for any operation o_i be t_i and the amortized cost be a_i. Hence, for any sequence of n operations we have

$$\sum_{i=1}^{n} a_i \leq cn. \tag{1}$$

Consider a sequence of n operations $o_1, o_2, \ldots, o_n$ such that any odd operation is *multipush*$\big((c+1), L\big)$ and any even operation is *multipop*$(c+1)$. Then the total actual cost for this sequence of operations is

$$T = \sum_{i=1}^{n} t_i = (c+1)n, \tag{2}$$

From (1) and (2), the main property of the amortized cost which is $\sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i$ does not hold. Therefore, our assumption on the amortized cost of these operations is wrong.

(b) It is required to model queue data structure with two stacks, such that amortized cost of operations is $O(1)$. Let us assume that we have two stacks S_1 and S_2. The simplest operation is *enqueue*(x), we model it with just simple call S_1.*push*(x), so we always put a new element on top of the first stack. Now, if we want to *dequeue*() and element, we face a problem, that the element which we need to return is on the bottom of the stack and there is no simple way to get it (if there are more than one element in the stack). To implement the *dequeue*() function, it is proposed to do the following. We put all elements that are currently in the S_1 stack (let it be k) into S_2 buy calling the $S_2.push(S_1.pop())$ k times, if S_2 is empty. Notice that the order of the elements in the S_2 stack is now the same as should have been in the queue, so after this costly operation time, we can model *dequeue* as a simple $S_2.pop()$ for the next k times and this operation is very cheap. Summing up, the idea is rather simple:

- *enqueue*$(\cdot)$: always *push* a new element on top of S_1.
- *dequeue*() : always *pop* an element from S_2.
- if S_2 is empty call $S_2.push(S_1.pop())$ until all elements of S_1 are in S_2. Return $S_2.pop()$.

Let us now take a closer look to the amortized cost of these operations. We will use **accounting method** of analysis. The real costs of operations that we need to pay are the following: putting an element into queue (on top of S_1) costs 1; if second stack is not empty getting an element costs 1; if S_2 is empty and S_1 has k elements the cost is $2k+1$, . Consider the following amortized cost of operations:

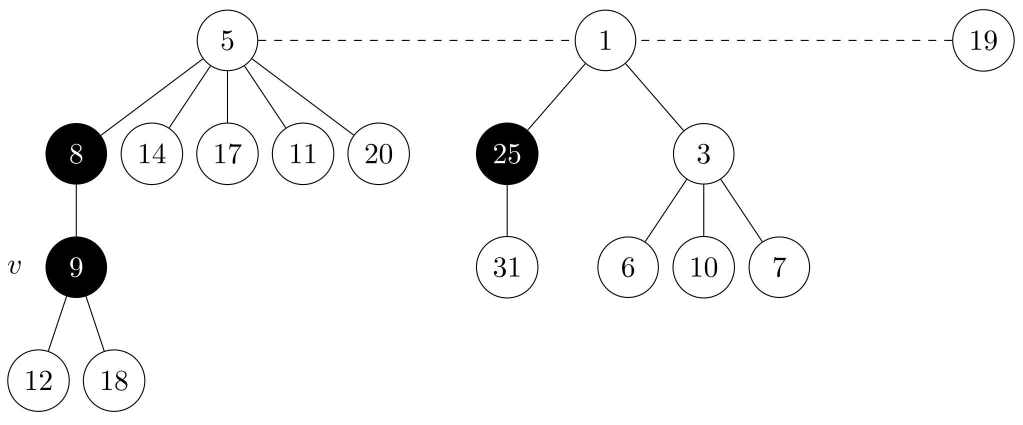
- *enqueue*$(\cdot)$: amortized cost is 3, from which we put 2 to account and 1 is the real cost.
- *dequeue*$(\cdot)$: amortized cost is 3. If stack S_2 is not empty, we delete the top element (real cost 1) and put 2 to the account. If stack S_2 is empty, we copy all elements (assume k) from S_1 to S_2 and take for each copied element 2 from the account (1 for each $S_1.pop()$ and $S_2.push()$, so $2k$ for copying) and also, we need to call $S_2.pop()$ one time to return the required element. There real cost is $2k+1$. Summing this up, we get amortized cost 3 if S_2 is not empty and we get $3 = 2k+1-2k+2$ in case if S_2 is empty.

Notice, that the number of elements that we move from S_1 to S_2 for any sequence of operations is always not more than number of *enqueue*$(\cdot)$ operation calls and so, for any execution: since

$$\sum_{i=0}^{n} t_i \leq \sum_{i=0}^{n} a_i = 3n$$

Exercise 2: Fibonacci Heaps (0.5+2.5+3 points)

(a) Consider the following Fibonacci heap (black nodes are marked, white nodes are unmarked). How does the given Fibonacci heap look after a *decrease-key*$(v, 2)$ operation and how does it look after a subsequent *delete-min* operation?

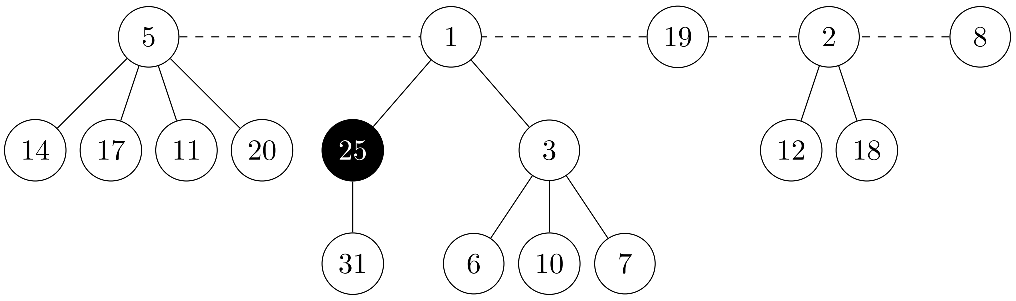


(b) Fibonacci heaps are only efficient in an amortized sense. The time to execute a single, individual operation can be large. Show that in the worst case, both the *delete-min* and the *decrease-key* operations can require time $\Omega(n)$ (for any heap size n).

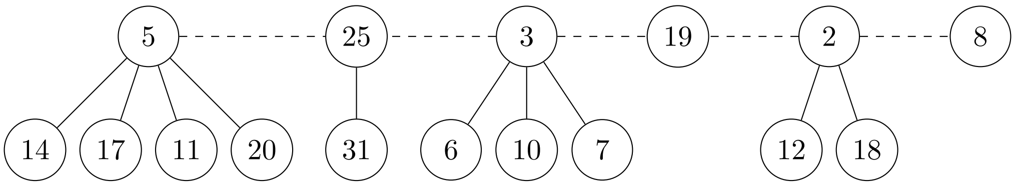
Hint: *Describe an execution in which there is a delete-min operation that requires linear time and describe an execution in which there is a decrease-key operation that requires linear time.*

Solution

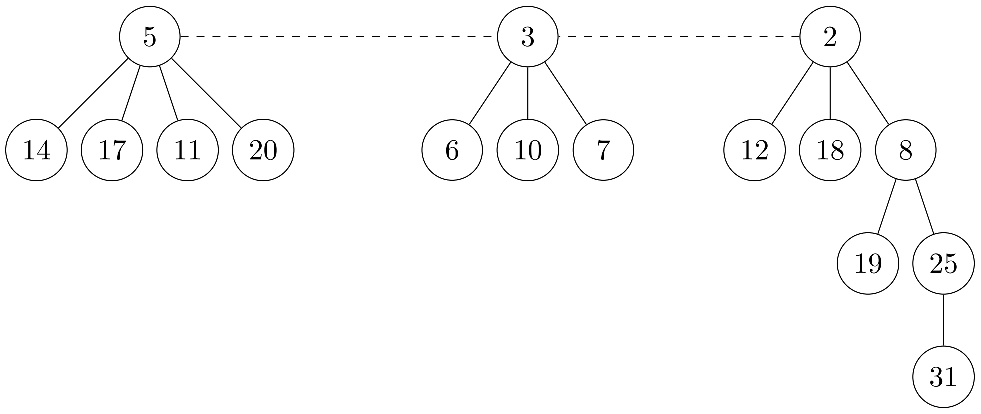
a) The *decrease-key* operation cuts the node – together with its subtree – off the tree and inserts it as a root to the root list, decreases the key and possibly updates the pointer to the minimum. The latter is not the case here. But then we are not done yet as its parent just lost a child and has to be marked. The parent of v however is already marked, which causes it to be cut as well, together with its remaining (empty) subtree, and being inserted into the root list, again checking whether the pointer to the minimum has to be updated. This goes on recursively until we reach the root or an unmarked node (in our case the node with key 5). Note that we can unmark any root[1]: if the root gets added to another tree, it loses the mark. If the root loses another child, we do not need to cut it off, as it is already in the root list.
Thus, the resulting heap looks like this:



If we delete the minimum, the node with key 1 gets removed and all its subtrees are inserted into the root list (we can unmark the roots of those subtrees, if we want).

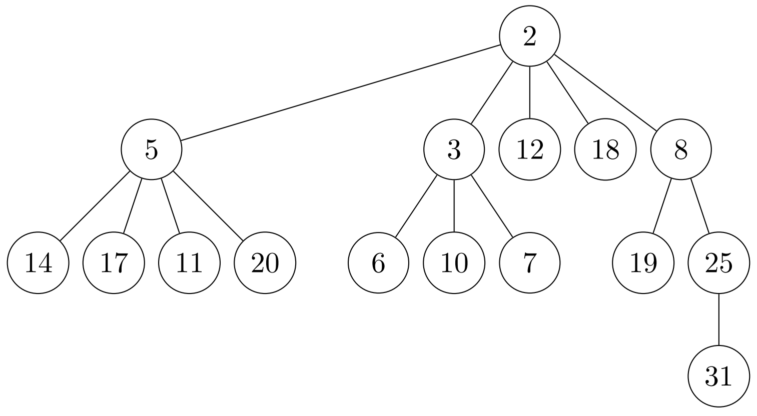


Following up is a *consolidate*, that merges trees of same rank. 8 merges with 19 and together they are a tree of rank 1. Which then merges with the tree rooted at 25 and is now a tree of rank 2. Like the one rooted at node v, now having the key 2. Let us merge those two trees, too. The current heap looks like this:

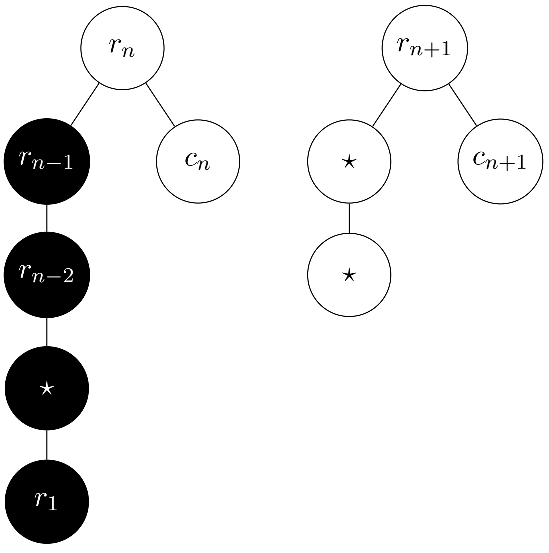


We have two trees of rank 3 and merge them, getting a tree of rank 4 rooted at v, which we then merge with the tree rooted at 5:

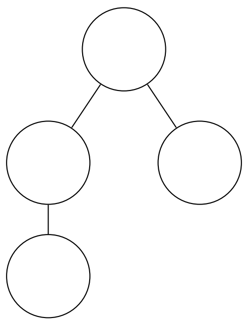
[1]The unmarking of a node in the root list has no effect on any future operation. Furthermore, the proof of the amortized costs works as it is because the potential function only considers marked nodes which are **not** in the root-list.



b) A costly *delete-min*:
First n elements are added to the heap, which causes them all to be roots in the root list. Deleting the minimum causes a *consolidate* call, which combines the remaining $n-1$ elements, which need at least $n-2$ *merge* operations, i.e., it costs $\Omega(n)$ time.
A costly *decrease-key* operation: (more difficult)
We construct a degenerated tree. Assume we already have a tree T_n in which the root r_n has two children r_{n-1} and c_n, where c_n is unmarked and r_{n-1} is marked and has a single child r_{n-2} that is also marked and has a single child r_{n-3} and so on, until we reach a (marked or unmarked) leaf r_1. In other words, T_n consists of a line of marked nodes, plus the root and one further unmarked child of the root. We give the root r_n some key k_n.
We now add another 5 nodes to the heap and delete the minimum of them, causing a *consolidate.* In more detail let us add a node r_{n+1} with key $k_{n+1} \in (0, k_n)$, one with key 0 and 3 with keys $k' \in (k_{n+1}, k_n)$. When we delete the minimum, first both pairs of singletons are combined to two trees of rank 1, which are combined again to one binomial tree of rank 2, with the node r_{n+1} as the root and we name its childless child c_{n+1} (confer the picture for the current state).



Since also T_n has rank 2 we now combine it with the new tree and r_{n+1} becomes the new root. We now decrease the key of c_n to 0 as well as the keys of the two unnamed nodes and delete the minimum after each such operation, as to cause no further effect from *consolidate.* Decreasing the key of c_n, however, will now mark its parent r_n, as it is not a root anymore. Thus the remaining heap is of exactly the same shape as T_n, except that its depth did increase by one: a T_{n+1}.
Can we create such trees? We sure can by starting with an empty heap, adding 5 nodes, deleting one, resulting in a tree of the following form:



We cut off the lowest leaf and now have a T_1. The rest follows via induction.
Obviously, a *decrease-key* operation on r_1 will cause a cascade of $\Omega(n)$ cuts if applied to a heap consisting of such a T_n.