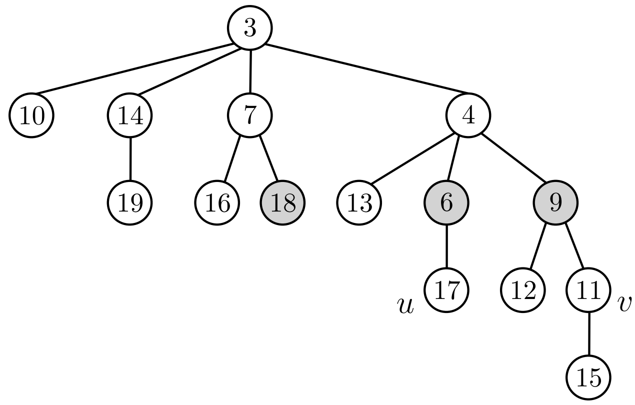
University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
P. Bamberger, P. Schneider

Algorithm Theory
Sample Solution Exercise Sheet 6

Due: Tuesday, 30th of November, 2021, 4 pm

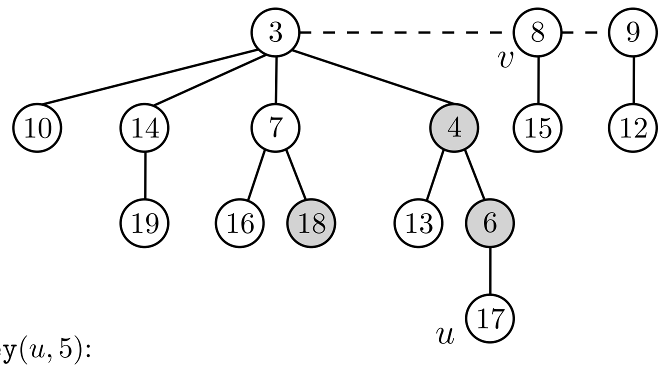
Exercise 1: Fibonacci Heap - Operations *(4 Points)*

Consider the following Fibonacci heap with marked nodes shown in gray and two dedicated nodes u, v. Give the state of the Fibonacci heap after conducting the operation `Decrease-Key`$(v, 8)$. Then conduct `Decrease-Key`$(u, 5)$ on the resulting Fibonacci heap and give the state of it.

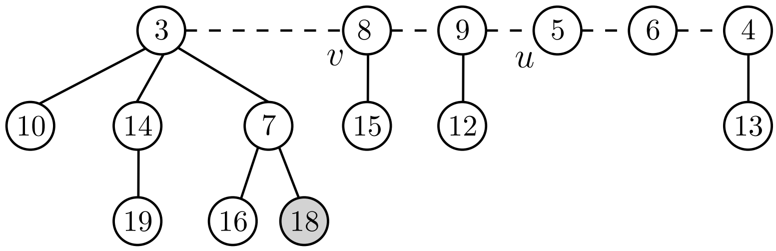


Sample Solution

After `Decrease-Key`$(v, 8)$:



After `Decrease-Key`$(u, 5)$:



Exercise 2: Fibonacci Heap - Questions *(6 Points)*

Suppose we "simplify" Fibonacci heaps such that we do *not* mark any nodes that have lost a child and consequentially also do *not* cut marked parents of a node that needs to be cut out due to a `decrease-key`-operation. Is the *amortized* running time

(a) ... of the `decrease-key`-operation still $\mathcal{O}(1)$? *(2 Points)*

(b) ... of the `delete-min`-operation still $\mathcal{O}(\log n)$? *(4 Points)*

Explain your answers.

Sample Solution

Two reasonable answers would be as follows.

(a) Yes. Not having to cut all your marked ancestor nodes only makes `decrease-key` faster. In fact each individual `decrease-key` operation has now runtime $\mathcal{O}(1)$. *(2 Points)*

(b) No. The reason is that we loose the recursive property that a given node with rank i has i children that have at least ranks $i-2, i-3, ...,$ respectively. This was required to show that each tree of a given rank has a *minimum size* of F_{i+2} (where F is the Fibonacci series) which grows exponential in i. Consequentially the maximum rank can not be too large, just $\mathcal{O}(\log n)$, as a tree with higher rank would require more than n nodes.

Now, if a node can loose an arbitrary number of children without being cut, the above property can not be guaranteed anymore. In particular, in extreme cases we could end up with a tree with rank $n-1$. Since `delete-min` has amortized runtime linear in the maximum rank, it will have a higher amortized running time (i.e., $\omega(\log n)$). *(4 Points)*

Exercise 3: Fibonacci Heap - Delete *(10 Points)*

We want to augment the Fibonacci heap data structure by adding an operation `delete(`v`)` to delete a node v (given by a direct pointer). The operation should have an amortized running time of $\mathcal{O}(\log n)$. Describe the operation `delete(`v`)` in sufficient detail and prove the correctness and amortized running time.

Remark: You can use the same potential function as for the standard Fibonacci heap data structure. Note however that after conducting `delete(`v`)` the Fibonacci heap must still be a list of heaps with maximum rank $D(n) \in O(\log n)$ and with a dedicated pointer to the minimum key.

Sample Solution

"Indirect" Implementation: Arguably, the simplest solution is to implement `delete(`v`)` using (a constant number of) our preexisting standard operations for the Fibinacci heap. For this we first execute a `decrease-key(`$v, -\infty$`)`, where we assume that $-\infty$ is special key smaller than every other key in the heap. Then we conduct a `delete-min`. Afterwards only the node v in the Fibonacci heap will be gone.

The correctness is clear. The actual running time $t_{\texttt{delete}}$ of `delete` is composed of that of the two operations $t_{\texttt{delete}} = t_{\texttt{decrease-key}} + t_{\texttt{delete-min}}$. We can trace the amortized running time $a_{\texttt{delete}}$ of the composed operation `delete` back to the sum of the two amortized runtimes of the single operations $a_{\texttt{decrease-key}}$ and $a_{\texttt{delete-min}}$ as well. Let Φ_{after} and Φ_{before} be the potential before and after executing `delete`. Then we have

$$\begin{aligned}
a_{\texttt{delete}} &= t_{\texttt{delete}} + \Phi_{\text{after}} - \Phi_{\text{before}} \\
&= t_{\texttt{delete-min}} + t_{\texttt{decrease-key}} + \Phi_{\text{after } \texttt{del-min}} - \Phi_{\text{before } \texttt{dec-key}} \\
&= t_{\texttt{delete-min}} + t_{\texttt{decrease-key}} + \Phi_{\text{after } \texttt{del-min}} - \Phi_{\text{after } \texttt{dec-key}} + \Phi_{\text{after } \texttt{dec-key}} - \Phi_{\text{before } \texttt{dec-key}} \\
&= t_{\texttt{delete-min}} + t_{\texttt{decrease-key}} + \Phi_{\text{after } \texttt{del-min}} - \Phi_{\text{before } \texttt{del-min}} + \Phi_{\text{after } \texttt{dec-key}} - \Phi_{\text{before } \texttt{dec-key}} \\
&= t_{\texttt{delete-min}} + \Phi_{\text{after } \texttt{del-min}} - \Phi_{\text{before } \texttt{del-min}} + t_{\texttt{decrease-key}} + \Phi_{\text{after } \texttt{dec-key}} - \Phi_{\text{before } \texttt{dec-key}} \\
&= a_{\texttt{delete-min}} + a_{\texttt{decrease-key}} \in \mathcal{O}(\log n).
\end{aligned}$$

"Direct" Implementation: We try to design the `delete(`v`)` operation to maintain the same conditions of the Fibonacci heap. Specifically, we ensure in the following that each node looses at most one rank by loosing a child.

We first cut out v and reinsert all child-heaps of v into the rootlist (not v itself). Since v's former parent now lost a child, we run the cascading cut procedure on v's former parent, meaning that all successive marked ancestors of v are cut out and reinserted into the rootlist. The closest previously unmarked ancestor of v is marked.

Finally we have to consider a special case that forces us to do another step. If the node v to be deleted is the current minimum, then we would have to go through the whole rootlist to find the new minimum.

Instead we run the consolidate routine like for `delete-min` (which also records the new min key). The reason for that is technical: we have to shrink the size of the rootlist R back down to $D(n)$ in order to "pay" that costly search (included in the consolidate) with the associated decrease in potential.

Runtime: The *actual cost* of our implementation of `delete(`v`)` is composed of the following components. The cutting and reinserting of the children of v can be done in actual $\mathcal{O}(1)$ using the linked list implementation. The next costly step is the cascading cuts procedure, which takes $\mathcal{O}(m_v)$ steps, where m_v is the number of successively marked ancestors of v.

Finally, let us assume that we delete the current minimum v. Then we have to consolidate, which takes time to the order of $\mathcal{O}(D(n) + |H.\texttt{rootlist}|)$, where $|H.\texttt{rootlist}|$ is the size of the rootlist and $D(n)$ the maximum rank of a Fibonacci heap of size n.

Overall, we have the true cost of $t_{\texttt{delete}} = m_v + D(n) + |H.\texttt{rootlist}|$. Note that we neglect constants which one can always adjust in the potential function to accommodate the true cost. The potential $\Phi = R + 2M$ (where R is the number of trees in the rootlist and m is the number of nodes) of the Fibonacci heap changes as follows:

$$\begin{aligned}
R_{\text{after}} &\leq R_{\text{before}} + D(n-1) - |H.\texttt{rootlist}| && \textit{now at most } D(n-1) \textit{ trees in rootlist} \\
M_{\text{after}} &\leq M_{\text{before}} - (m_v - 1) && m_v \textit{ ancestors loose marks, but one mark might be added back} \\
\Phi_{\text{after}} &\leq \Phi_{\text{before}} + D(n-1) - |H.\texttt{rootlist}| - 2(m_v - 1).
\end{aligned}$$

The difference $\Phi_{\text{after}} - \Phi_{\text{before}}$ can be used to offset our true costs $t_{\texttt{delete}} = m_v + D(n) + |H.\texttt{rootlist}|$:

$$\begin{aligned}
a_{\texttt{delete}} &= t_{\texttt{delete}} + \Phi_{\text{after}} - \Phi_{\text{before}} \\
&= m_v + D(n) + |H.\texttt{rootlist}| + \Phi_{\text{after}} - \Phi_{\text{before}} \\
&\leq D(n) + t_2 + |H.\texttt{rootlist}| + D(n-1) - |H.\texttt{rootlist}| - 2(m_v - 1) \\
&\leq D(n) + D(n-1) \in \mathcal{O}(\log n).
\end{aligned}$$