Exercise 1: Fibonacci Heap Worst Case  

(a) Consider the Fibonacci heap given below. Perform a delete-min operation. Give the state before and after the consolidate operation. Conduct the link operations exactly in the order in which the algorithm given in the lecture does it (Chapter 5 Part IV Slide 16). (4 Points)

(b) Give a valid instance of a Fibonacci heap where delete-min has a worst case runtime of \( \Omega(n) \) and explain why this is the case for that instance. (3 Points)

Sample Solution

(a) After delete-min, before consolidate

After consolidate
(b) Consider inserting the keys $1, \ldots, n$ into an empty Fibonacci heap. These are going to be inserted, in a lazy fashion, as heaps of rank zero into the rootlist. Then we run delete-min, which removes the heap with key 1 and triggers a consolidate. The consolidate routine iterates over the root-list, which still contains $n-1$ heaps. Therefore the total runtime of consolidate and consequentially of delete-min is $\Omega(n)$.

Exercise 2: Fibonacci Heap Modifications

(a) Assume that operation decrease-key never occurs. Show that in this case, the maximum rank $D(n)$ of a Fibonacci heap is at most $\lfloor \log_2(n) \rfloor$. (5 Points)

(b) We want to augment the Fibonacci heap data structure by adding an operation increase-key($v, k$) to increase the key of a node $v$ (given by a direct pointer) to the value $k$. The operation should have an amortized running time of $O(\log n)$. Describe the operation increase-key($v, k$) in sufficient detail and prove the correctness and amortized running time. (8 Points)

Remark: You can use the same potential function as for the standard Fibonacci heap data structure. Note however that after conducting increase-key($v, k$) the Fibonacci heap must still be a list of heaps, where the maximum rank $D(n) \in O(\log n)$.

Sample Solution

(a) First we show inductively that when there are no decrease-key operations, then a heap of rank $i$ in the rootlist has exactly $2^i$ nodes (we call the number of nodes the size of the heap in the following). A heap of rank 0 is just a single node, thus it has size $2^0 = 1$.

Given a heap $h$ of rank $i > 0$ which might also be a sub-heap attached to some parent node. The only way the degree $i$ of $h$ can be created is by linking two heaps $h_1, h_2$ of rank $i-1$. By induction hypothesis heaps of rank $i-1$ have $2^{i-1}$ elements. Therefore, the size of the heap $h = \text{link}(h_1, h_2)$ is the sum of the sizes of $h_1, h_2$, i.e., $2^{i-1} + 2^{i-1} = 2^i$.

Remark: When we execute a delete-min operation, then smaller heaps that are attached to the current minimum are cut and reinserted into the rootlist. But this does not change the the form of the subheaps of the root in any way, so the induction argument above remains valid.

Since we have only $n$ nodes in the Fibonacci heap in total, the heap with the biggest rank $D(n)$ must fulfill the inequality

$$2^{D(n)} \leq n \iff D(n) \leq \log_2 n.$$ 

Since $D(n)$ is an integer value we have $D(n) \leq \lfloor \log_2 n \rfloor$.

(b) Implementation and correctness arguments: As suggested in the remark, we try to design the increase-key($v, k$) 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. First we assert that for increase-key($v, k$) the new key $k$ is larger than the current key of $v$. If $k$ is smaller than or equal to all the keys of its child nodes, the heap condition is not violated after changing the key to $k$ and we do nothing else.

Otherwise we first cut out and reinsert all child-heaps of $v$ into the rootlist. Since $v$ has lost too many children (each node can loose at most one) we also cut $v$ from its parent and reinsert it as single node into the rootlist. 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 $v$ whose key we increased is the current minimum, then we have to go through the whole rootlist to find the new minimum (or to confirm that $v$ is still the minimum). But then we also have to run a consolidate like for delete-min. 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 (and consolidate) with the associated decrease in potential.

**Runtime:** The actual cost of our implementation of $\text{increase-key}(v, k)$ is composed of the following components. We have $t_1 \leq D(n)$ steps for cutting and reinserting all child-heaps of $v$, since $D(n)$ is the maximum number of children $v$ can have.

The next costly step is the cascading cuts procedure, which takes $t_2$, where $t_2$ is the number of successively marked ancestors of $v$ plus one or, alternatively, the distance to the closest unmarked ancestor of $v$.

Finally, let us assume that we actually increase the key of current the minimum $v$. Then we have to find a new minimum and also consolidate, which takes time to the order of $t_3 \leq |H.\text{rootlist}|$, where $R$ is the size of the rootlist.

The potential of the Fibonacci heap changes as follows:

$$
R_{\text{new}} = R_{\text{old}} + D(n) + 1 - |H.\text{rootlist}|
$$

$$
M_{\text{new}} = M_{\text{old}} - (t_2 - 1)
$$

$$
\Phi_{\text{new}} = \Phi_{\text{old}} + D(n) + 1 - |H.\text{rootlist}| - 2(t_2 - 1).
$$

The difference $\Phi_{\text{new}} - \Phi_{\text{old}}$ can be used to offset or more precisely amortise our true costs $t_1 + t_2 + t_3$:

$$
a_i = t_1 + t_2 + t_3 + \Phi_{\text{new}} - \Phi_{\text{old}}
\leq D(n) + t_2 + |H.\text{rootlist}| + \Phi_{\text{new}} - \Phi_{\text{old}}
= D(n) + t_2 + |H.\text{rootlist}| + D(n) + 1 - |H.\text{rootlist}| - 2(t_2 - 1)
= 2D(n) + 3 - t_2
\leq 2D(n) + 3 \in O(\log n).
$$