1. We rebuild the subtree by a divide-and-conquer algorithm (put the middle element at the root and construct the left and right subtrees recursively). Therefore, the tree can have a size difference of at most 1 between the two subtrees after the rebuilding.

Since
\[ I(x) = \max\{0, |\text{size}(\text{left}(x)) - \text{size}(\text{right}(x))| - 1\}, \]
the imbalance of the rebuilt subtree is 0. Recursively applying this to each level, the remainder of the entire tree has 0 imbalance, i.e., the new tree has imbalance of 0.

2. For deletion: If no rebuilding is needed, the analysis is the same as in the lecture notes, because the imbalance is not changed. If rebuilding is needed, the actual cost is still \( O(\log n) + \Theta(n) \). The imbalance after rebuilding could be at most \( \Theta(n) \) (instead of 0 in the original version) and
\[
\Delta \Phi \leq \hat{c} \text{size}(T)/2 + \Theta(n)
\]
By scaling \( \hat{c} \), the \( \Theta(n) \) in both actual cost and potential change can be canceled and the amortized cost is \( O(\log n) \).

For insertion: If no rebuilding is needed, then the imbalance change of path from the root to the new node is at most \( \text{height}(T) \leq \beta \log n \). Since the number of marked nodes does not change, we have
\[
\Delta \Phi \leq \beta \log n
\]
The actual cost is simply a depth search which costs \( O(\log n) \). So, the amortized cost is \( O(\log n) \).

Now, if the tree is rebuilt, the actual cost is \( O(\log n) + \Theta(\text{size}(x)) \). The analysis of potential change from the lecture notes applies to this problem as well, except for the last step on page 4 which is now changed to
\[
I(x) = \text{size}(\text{left}(x)) - \text{size}(\text{right}(x)) \geq \frac{\text{size}(x)}{2^{1/\beta}} - \left[1 - \frac{1}{2^{1/\beta}}\right]\text{size}(x) - 1
\]
\[
= (2^{1-1/\beta} - 1)\text{size}(x) + 1
\]
\[
> (2^{1-1/\beta} - 1)\text{size}(x)
\]
\[
= \Theta(\text{size}(x))
\]
and after rebuilding \( I(x) \leq 1 \). Similar to before, the imbalance of each ancestor of \( x \) increases by at most 1, which increases the potential at most \( O(\log n) \). Therefore, the potential change
\[
\Delta \Phi \leq -\Theta(\text{size}(x)) + \Theta(\log n)
\]
By choosing the right \( c \), we can cancel the terms \( \Theta(\text{size}(x)) \) and arrive at an amortized cost is \( O(\log n) \).

3. PhD Problem 5: If there is no rebuilding needed, analysis on potential remains the same as in the lecture notes. So the amortized cost in this case is \( O(\log n) \). If some subtree is rebuilt, searching for an unbalanced node and rebuilding both take \( O(\text{size}(x)) \) according to the lecture notes (let’s suppose...
$x$ is the lowest unbalanced node along the path from the root to the newly inserted item). A simple post-order traversal would be able to find the size and height information of $x$. So the actual cost is higher than the original because of searching for the unbalanced node and calculating the height and the size, but still in the order of $O(\text{size}(x))$ as before. Potential analysis remains the same. Therefore the amortized cost is still $O(\log n)$.

4. The basic idea is to build the Fibonacci heap recursively: first build a Fibonacci heap that is a linear chain of size $n - 1$ and then do a few operations to add one more node to the chain. Specifically, suppose we have built a chain-like Fibonacci heap $F$ of size $n - 1$ with the minimum key $m + 1$ and now we want to add one more node. What we do is insert the elements $m$, $m + 2$ and $m - 1$ to the heap, extract the minimum node (what happens here: $m - 1$ is deleted, $m$, $m + 2$ are consolidated and then they are further consolidated with the chain), then decrease the key of the second child (if any) of $F$.min to $m - 1000$, and extract the minimum node again (to remove the second branch of the tree). We now obtain a chain-like Fibonacci heap of size $n$ with the minimum key $m$. The algorithm is presented in Algorithm 1.

Algorithm 1 Build-Linear-Fibonacci-Heap($n, m$)

1: if $n = 1$ then
2: return a Fibonacci heap with only one element $m + n - 1$
3: if $n = 2$ then
4: return a Fibonacci heap with only two nodes: an element $m + n - 2$ plus its child $m + n - 1$
5: $F =$Build-Linear-Fibonacci-Heap($n, m$)
6: Insert($F, m$)
7: Insert($F, m + 2$)
8: Insert($F, m - 1$)
9: Extract-min($F$)
10: Let $c$ be the second child of $F$.min
11: Decrease-key($F, c, m - 1000$)
12: Extract-min($F$)
13: return $F$

5. PhD Problem 6: Note that, as long as $D(n)$ is bounded, for the following 5 procedures: MAKE-HEAP, INSERT, MINIMUM, UNION and DECREASE-KEY, amortized costs remain the same as the original cost $O(1)$. What might be changed is for EXTRACT-MIN and DELETE. Well show that for any $k$, as long as $k$ is a constant and independent of $n$, $D(n)$ is still bounded by $O(\log n)$ but with a different constant factor compared to the original Fibonacci Heap. Let $s_i$ be the minimum number of nodes of degree $i$. According to the analysis on page 525 of the text book, when $d \geq k$ we have

$$s_d = s_{d-1} + s_{d-k}$$

We can prove by induction that

$$s_d \geq (1 + \delta)^{d-k}$$

where $\delta$ is a positive number. Based on the textbook and the recurrence, we have

$$\text{size}(x) \geq s_d$$

$$\geq k + \sum_{i=k}^d s_{i-k}$$
Note that $s_d \geq (1 + \delta)^{d-k}$ for all nonnegative integers $dk$. When $d = 1, 2, \cdots, k1$, $s_d = 1$. Thus, we have $n \geq \text{size}(x) \geq s_d \geq (1 + \delta)^{d-k}$, which implies that $k \leq \log_{1+\delta}(n)$. Therefore, the maximum degree $D(n)$ of any node in this Fibonacci heap is $O(\log n)$ when $k$ is constant. The complexity is the same as the original Fibonacci heap in that the maximum degree $D(n)$ is bounded by $O(\log n)$.

6. **Fib-Heap-Change-Key**: Suppose the original key is $k_0$ and the new key is $k$. There are two cases.

1) If $k_0 < k$, we delete the original node and then insert an new element with key $k$ to the heap. The amortized cost of deletion and insertion is $O(\log n)$ and $O(1)$ respectively, so the total cost is $O(\log n)$.

2) If $k_0 \geq k$, we can just call $\text{decrease-key}$ function to do it, whose amortized cost is $O(1)$. Therefore, the amortized cost of Fib-Heap-Change-Key is $O(\log n)$.

**Fib-Heap-Prune**: First, perform postorder traversal on all the trees in the heap (which costs $O(n)$) and store the ordering of nodes. Then the algorithm deletes nodes by this order, so that it’s always deleting leaves and there is no need to rearrange the heap. We don’t need to do cascade-cut every time a leaf is removed since its parent might be removed soon; instead, cascade-cut is done only once after $q$ nodes are all removed. The cost of deleting each node is $O(1)$. The cost of cascade-cut is $O(\log n)$. Thus, the total actual cost is $O(n) + q \times O(1) + O(\log n) = O(n)$. We still use the same potential function. Every time we delete a node, the potential change is at most 1, so the total potential change is $O(q)$. The amortized cost of prune is $O(n) + O(q) = O(n)$. 