1. Suppose that, instead of splaying every time we locate a node, we only splay every other time (i.e., on the second, fourth, sixth, eighth, etc. operations). In a splay tree with \( n \) nodes, show that this modification could lead to a \( \Theta(mn) \) running time for a sequence of \( m \) operations in the worst case. You are free to choose the initial tree structure and sequence of operations; please state what they are in your answer. Remember to show both an upper and lower bound on the worst case complexity.

**Answer:** Suppose that the initial structure is a chain of nodes, where each node is the left-child of its parent (except for the root, which has no parent). Let \( x_1 \) refer to the smallest node (which is deepest in the tree) and \( x_n \) refer to the largest node (which is the root). We obtain a running time of \( \Theta(mn) \) by accessing \( x_1 \) on the odd accesses (1, 3, 5, etc.) and \( x_n \) on the even accesses (2, 4, 6, etc.). For the odd accesses, finding \( x_1 \) takes \( \Theta(n) \) time since we must descent to a depth of \( n \) in order to locate \( x_1 \). Since we only splay on even accesses, this does not change the structure. For the even accesses, finding \( x_n \) takes constant time, as does splaying it (since \( x_n \) is already the root). This does not change the structure either. Therefore, we can repeat these two operations an arbitrary number of times. The “odd” operations each take \( \Theta(n) \) time, and the “even” operations each take \( \Theta(1) \). For a sequence of \( m \) accesses, there will be at least \( m/2 \) odd operations which will take a total of \( \Theta(mn/2) \), which is \( \Theta(mn) \).

2. In a splay tree with \( n \) nodes, what is the largest possible increase in potential that can be created from a single rotation (not a splay step)? Describe what the splay tree looks like before and after this worst-case single rotation.

**NOTE:** Please use the potential function we defined in class, \( \Phi = \sum_x r(x) = \sum_x \lg(\text{size}(x)) \).

**Answer:** Let \( y \) be the parent before the rotation and let \( x \) be the new parent, after the rotation. In any rotation, the old parent \( y \) only decreases in rank, since it is no longer an ancestor of \( x \) or \( x \)’s right subtree. \( x \) increases in rank to be equal to \( y \)’s old rank. The ranks of all other nodes are unchanged. Let \( r' \) be the new rank after the rotation and \( r \) be the old rank before the rotation. The change in potential from a single rotation is always:

\[
r'(x) + r'(y) - r(x) - r(y) = r'(y) - r(x) \quad (\text{since } r(y) = r'(x)).
\]

To maximize this value, we want the largest possible \( r'(y) \) and the smallest possible \( r(x) \). A lower bound for \( r(x) \) is clearly 0, since \( \text{size}(x) \geq 1 \). An upper bound on \( r'(y) \) is clearly \( \lg(n - 1) \), when \( y \) is an ancestor of all nodes except for its new parent, \( x \). This would lead to an increase of \( \lg(n - 1) - 0 = \lg(n - 1) \).

We can obtain this increase in the following situation: Before the rotation, the root of the tree has a right subtree with 1 node and a left subtree with all remaining \( n - 2 \) nodes (in any fixed configuration). Rotating the root’s right child up to the root yields a tree where the new root has no right child. The left child of the new root has \( n - 2 \) nodes in its left subtree and zero nodes in its right subtree. Here, \( r'(y) = \lg(n - 1) \) and \( r(x) = 0 \), proving that our upper bound in the previous paragraph is tight.
3. Suppose that we want to make Fibonacci heaps even lazier by skipping \textit{consolidate} when the number of trees in the root list, \(t\), is less than some constant \(k\). For \(k \leq 2\), this is clearly identical to a regular Fibonacci heap. For larger values of \(k\), however, we may run \textit{consolidate} less often than in a regular Fibonacci heap.

(a) Explain how to implement \textit{delete-min} in this modified data structure along with its worst-case running time. (You may assume \textit{consolidate} is available to use as a subroutine.)

(b) Prove that the amortized running time of \textit{delete-min} is still \(O(\log n)\). You may assume any results proven in class. (Hint: Use the same potential function that we used in the analysis of regular Fibonacci heaps so that you only have to analyze how the new \textit{delete-min} is different.)

(c) (Grads only) Determine the amortized running time of \textit{delete-min} when \(k\) is replaced with an arbitrary function on the number of nodes, \(f(n)\). (For example, when \(f(n) = \lg n\), then \textit{consolidate} is only performed when there are at least \(\lg n\) trees in the root list.) Express the new amortized running time in terms of \(n\) and \(f(n)\).

\textbf{Answer:}

a. First, we remove the minimum node and add its children to the root list. This takes at most \(O(\log n)\) time, since \(D(n)\) is still \(O(\log n)\). (The lemma that gives a minimum bound on the size of a tree with \(k\) children is unaffected by consolidating less often.) Next, we check to see if the number of trees in the root list is less than \(k\). This takes \(O(k)\) time, since if the tree has more than or equal to \(k\) trees, we can stop once we’ve seen \(k\) of them. If the root list has more than \(k\) trees, we perform a regular consolidate. In the worst case, all \(n\) nodes are present in the root list, so the operation takes \(O(n)\) time. If the root list has fewer than \(k\) trees, then we update the minimum pointer to the minimum of the \(k\) tree roots. This takes at most \(O(\min(n, k))\) time. Therefore, the worst-case total time is \(O(n)\).

b. (We use the same potential function as in class, which is initially zero and never negative, so in a sequence of operations, our amortized costs are an upper bound on the actual costs.) There are two cases to consider. When the number of trees in the root list is less than \(k\), the amortized cost is:

\[
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \\
\leq (D(n) + k) + D(n) \\
= O(\log n)
\]

The real cost is \(D(n)\) to add children to the root list and \(k\) to find the new minimum. The change in the potential is at most an increase of \(D(n)\) in the number of trees in the root list. Since \(k\) is a constant, this can be reduced to \(O(\log n)\).

If the number of trees in the root list is more than \(k\), then the amortized cost is the same as proven in class: \(O(\log n)\).

In either case, the total amortized cost is \(O(\log n)\).
c. Suppose the minimum threshold is $f(n)$. The total amortized time is:

$$
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}
\leq (D(n) + f(n)) + D(n)
= O(\log n + f(n))
$$