Problem 2

We can construct a binomial heap given an order k and minimum number of nodes m by inserting $2^{\lceil \log_2(m+1) \rceil}$ nodes and then calling extract-min once to consolidate the trees. This results in a heap with $2^{\lceil \log_2(m+1) \rceil} - 1$ nodes that has child subtrees of order $0 \ldots \lceil \log_2(m+1) \rceil - 1$. Since the number of nodes is $2^{\lceil \log_2(m+1) \rceil - 1} \geq m$ nodes. We can then find the minimum key of the tree, s and call decrease-key on the roots of the child trees starting from the tree of order 0 for $k-1$ times, setting the root of the tree to $s-1$. Since $s-1$ is guaranteed to be lower than the root of the tree, this will cause the children binomial heaps to be cut off and increase the order of the overall binomial heap by one every time. We can repeat this $k-1$ times until we have a binomial heap of order k with more than m nodes, as desired.

Problem 3

We begin with a lazy binomial heap and perform modifications in order to achieve our desired amortized runtime of new-pq, insert, find-min, meld, and add-to-all in amortized time $O(1)$ and extract-min in $O(\log n)$. We will design around the function add-to-all since it is the major addition to our data structure.

In order to achieve amortized constant time melds, we also introduce a separate, auxiliary binomial heap that will keep track of the minimum element of the heap at all times. For clarity, we will refer to this auxiliary heap as A and the overall structure that we are building as B. Operations on A are standard lazy binomial heap operations as outlined in lecture. The roots of the binomial trees are stored in a linked list (one for A and one for B).

We introduce the notion of an augmented property d on each node of B. This field is an integer that represents the add-to-all values that have been added to the priority of the node and all its children. In addition, we keep a counter c with an initial value of 0 in B to keep track of add-to-all's that have been performed to B. For example, a node (with no parents) with $d = 3$ and priority 2 has an effective priority of $3+2+2 = 7$ if c is 2 and its children's priorities are also effectively $d+c = 3+2$ higher.

A description of each of the priority queue operations is below. Let $ROOT(x)$ be the root of a binomial heap x.

These operations assume that they are being performed on a data structure B with auxiliary heap A.

new-pq: Initialize an empty binomial heap for A and B. No nodes are on either heap.

insert(v, k): Create a node n_1 with value v, key k, and aux value $d = -c$ (since the effective priority of every element in B is actually $k + d + c$). Append this element to the end of the linked list of roots in B. Also create a new node n_2 with key equal to $k + d$ and value v as a pointer to n_1. Insert n_2 into A.

find-min: Perform find-min on A to get the element with the lowest key, n. Follow the value v of n (which is a pointer to a node in B) and return that node.

meld(B, pq_2): Append $ROOT(pq_2)$ to the linked list of root nodes in B. Subtract c from $ROOT(pq_2).d$ to account for the counter. Create a new node n with key $n.k = ROOT(pq_2).k + ROOT(pq_2).d$ and value v as a pointer to $ROOT(pq_2)$. Insert n into A and return B as the new tree.

add-to-all(s): Set c to $c + s$.

extract-min:
Phase 1: Extraction. Find the minimum element n_a in A and follow the pointer at $n.v$ to get the node n_b in B. Extract out the immediate children of $ROOT(n_b)$ and add $ROOT(n_b).d$ to the children's auxiliary values (i.e. increment/decrement the axuiliary priority of the children by the auxiliary priority of the root). Take these children and append them to the linked list of root nodes for B. Clear the auxiliary heap A in preparation for changes to B.

Phase 2: Merge. We can now merge the trees that are in the linked list as demonstated in lecture 8: by creating space for $O(\log n)$ trees and then assigning and merging trees based on their size (removing them from the linked list), a standard lazy binomial operation. When the merging in the binomial heap occurs, we have two trees: T_1 and T_2 that must be merged. Without loss of generality, let T_1 have a lower $k + d$ value. We must maintain accuracy of the d fields stored in the node, so we set $T_2.d = T_2.d - T_1.d$ and let T_1 be the root of the new merged tree. This propagation occurs on every tree merge operation. At the end of **extract-min** and its merging, there will be only $O(\log n)$ trees. After merge operations, go through the trees remaining and add c to the root nodes' d field.

Lastly, set $c = 0$ again and repopulate A by creating nodes as before for the root nodes remaining in the linked list in B. These nodes should have pointer values pointing to the node in B and keys equal to $k + d$ of the node in B.

This series of operations allows us to maintain the node with the lowest priority at the top of the heap while offering operations such as add-to-all and meld in amortized constant time. Since the counter values and auxiliary d values are propagated to the children every time the minimum node is extracted/replaced, we maintain correct functionality.

Proof. of Runtime In order to prove amortized runtime, we use a potential function $\phi = \#$ trees in $A + \#$ trees in B.
new-pq: $O(1)$ actual. No potential change. $O(1)$.

insert(v, k): $O(1)$ actual. $\Delta\phi = 2$ since one node is added to B and A each. Amortized cost: $O(1) + O(1) \cdot 2 = O(1)$.

find-min: $O(1)$ actual. $\Delta\phi = 0$ since no trees are added.

meld(B, pq_2): $O(1)$ actual (add to linked list, subtract counter). $\Delta\phi = 2$ since one node is added to B and one to A.

add-to-all(s): $O(1)$ actual. $\Delta\phi = 0$ since no trees are added.

extract-min:
actual $O(1)$ (find minimum) + $O(\log n)$ (extract and manipulate $O(\log n)$ children, append these children to root node, add c to root afterwards, repopulate A) + T (merge trees). $\Delta\phi = -T + O(\log n)$ (remove T trees, create O$(\log n)$ of them). Amortized: $O(1) + O(\log n) + T + O(1) \cdot (-T + O(\log n)) = O(\log n)$ ■