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 melded queues and the minimum elements of those queues at all times. For clarity, we will refer to this auxiliary heap as A and the overall structure that we are building as B. The nodes in A will have keys representing the value of the minimum element of each queue that was melded to B and values that are pointers to the elements themselves. Operations on A are standard lazy binomial heap operations as outlined in lecture.

We introduce the notion of an augmented property d on each node of B that starts initially at 0. This field is an integer that represents the add-to-all Δk values that have been added to the priority of the node and all its children over its lifetime. 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 overall. 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 for the queue that it is in and its children's priorities are also effectively $d + c = 3 + 2$ higher.

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. Update the min pointer (as in a normal lazy binomial heap) for B if the value $k+d$ of the new node is less than the $k+d$ node that the min pointer is pointing at.

find-min: Perform find-min on A to get the node with the lowest key, n. Compare this key to the $k + d$ of the node pointed to by the min pointer and return the node with the lower value..

meld(B, pq_2): Insert the minimum element of pq_2 (using find-min to get) into A. Subtract c from $pq_2.c$ to account for the counter. 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 compare it to the node pointed to by the min pointer (as in find-min). Let this node be n. Take the root trees of the queue that contains n (this information is stored in A as well) and add $n.d$ to the children's auxiliary values (d-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 in B. For each of the melded queues in A, add its root trees to the linked list in B as well (excluding the tree that has n as the root if n was taken from A). A can then be cleared.

Phase 2: Merge. We can now merge the trees that are in the linked list of B 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 occurs on every tree merge operation to preserve the accuracy of d at the root. At the end of **extract-min** and its merging, there will be only $O(\log n)$ trees.

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. Since everything in B is subject to a shift by c, we can do add-to-all easily as well.

Proof. of Runtime In order to prove amortized runtime, we use a potential function $\phi = \#$ nodes in $A + \#$ trees in B. Let T_A be the number of nodes in A and T_B be the number of root trees in B. The potential function thus becomes $\phi = T_A + T_B$
new-pq: $O(1)$ actual. No potential change. $O(1)$.

insert(v, k): $O(1)$ actual (adding to linked list). $\Delta\phi = 1$ since one node is added to B as a new tree. Amortized cost: $O(1) + O(1) = O(1)$.

find-min: $O(1)$ actual (since A is a lazy binomial heap, we can use $O(1)$ here as "actual" even though part of it is amortized). $\Delta\phi = 0$ since no trees are added and A is not changed.

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

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

extract-min:

Actual $O(1)$ (find minimum) + $O(\log n)$ (exposing children on extract and adding d to children aux values) + $O(\#$ trees in$B)$ (merge trees) + $O(\#$ nodes in$A)$ (merge melded heaps). $\Delta\phi = -T_A - T_B + O(\log n)$ (remove T_A nodes from A, remove T_B trees from B, create O$(\log n)$ of them after merging).

Amortized: $O(1) + O(\log n) + O(T_A) + O(T_B) + O(1) \cdot (-T_A - T_B + O(\log n)) = O(1) + O(\log n) + O(\log n) = O(\log n)$ ■