Problem 17.2

(a) There are $O(\log n)$ arrays and we need to query each of them. So the total query time is $O(\log^2 n)$. Although the query time decreases as the array size gets smaller, but that doesn't change the asymptotic bound.

(b) Recall that merging two sorted arrays into one takes time linear to the total size of the two arrays. We first merge the new element with the smallest array of size 2^0, resulting in an array of size 2^1. Then we merge it with the array of size 2^1, resulting in an array of size 2^2. Then we merge it with the array of size 2^2, etc. The total cost is thus $2^1 + 2^2 + 2^3 + \cdots + 2^{i+1} = O(2^i)$.

(c) We can use the aggregate method. Assume n is a power of 2. Over all n insertions, the array of size n is built once, the array of size $n/2$ is built 2 times, the array of size $n/4$ is built 4 times, etc. So the total cost is $n \cdot 1 + n/2 \cdot 2 + \cdots + 1 \cdot n = O(n \log n)$.

We can also use the accounting method. We give $O(\log n)$ credits to each element when it is inserted. Observe that elements only move from a smaller array to a bigger array, so it is moved $O(\log n)$ times. Every time we build an array, we charge the cost to all the elements involved, which is $O(1)$ per element. So the $O(\log n)$ credits are enough to pay for all the costs.

Problem 17.3

(a) We first do an in-order traversal of the tree to collect all nodes in sorted order and store them in an array. Then we use divide-and-conquer to rebuild the subtree. We pick the element in the center of the array as the root, and then recursively build the left subtree on the left half of the array, and the right subtree on the right half of the array. Let $T(n)$ be the running time of this algorithm on an array of size n, then we have $T(n) = 2T(n/2)+1$, which solves to $T(n) = O(n)$.

(b) First, observe that the height of the tree is $O(\log n)$, because the subtree size reduces to a constant fraction of α for each level.

We use the potential method. Define the potential of the tree to be

$$\Phi(T) = \sum_{u \in T} \max\{|size(u.left) - size(u.right)| - 1, 0\}.$$

An insertion adds 1 to $size(u)$ for each u that is an ancestor of the inserted node. This causes the potential to increase by at most $O(\log n)$. Thus, an amortized cost of $O(\log n)$ is enough to pay for both the actual cost of the insertion and the potential increase. The same holds for a deletion. Now consider a partial rebuild at node u, the cost of which is $O(size(u))$. Before the rebuild, we have

$$|size(u.left) - size(u.right)| - 1 > (\alpha - (1 - \alpha))size(u) - 1 = (2\alpha - 1)size(u) - 1.$$

After the rebuild, the potential in the entire subtree below u is 0. So the potential drops by at least $(2\alpha - 1)size(u) - 1$. If $size(u) > 1/(2\alpha - 1)$, this is $\Omega(size(u))$, so it is enough to cover the rebuilding cost, if we scale up the potential function by a sufficiently large constant. If $size(u) < 1/(2\alpha - 1)$, we simply charge this cost to the insertion/deletion that causes this partial rebuild, which increases the amortized cost by only $O(1)$.

Problem A This means that we should always have $\alpha > \frac{1}{1+\epsilon}$. So we use the following strategy (there can be other strategies): When $\alpha = 1$, increase the table size by a factor of $1+\epsilon/2$, which makes $\alpha = \frac{1}{1+\epsilon/2}$. When $\alpha = \frac{1}{1+\epsilon}$, we reduce the table size to a fraction of $\frac{1}{1+\epsilon/2}$, which makes $\alpha = \frac{1/(1+\epsilon)}{1/(1+\epsilon/2)} = \frac{1+\epsilon/2}{1+\epsilon}$. Suppose the table has size m after a rebuild. In either case, we see that it takes at least $\Omega(\epsilon m)$ operations before reaching $\alpha = 1$ or $\alpha = \frac{1}{1+\epsilon}$ again. So by the same argument in the lecture notes, the amortized cost is $O(1/\epsilon)$.