17-2 :  
(a) Perform binary search on each of the $k$ arrays, starting with $A_{k-1}$ down to $A_0$. Stop as soon as the key is found. This algorithm is clearly $O\left(\log^2 n\right)$. For infinitely many values of $n$, this upper bound is tight. Consider the case where $n = 2^m - 1$ for some integer $m$, and we are asked to find an element which lies in $A_0$. In this case all $k$ arrays are full and the algorithm performs binary search on all of them. The run time is thus:

$$
\sum_{i=0}^{k-1} \log 2^i = \sum_{i=0}^{k-1} i = \Omega(k^2) = \Omega(\log^2 n)
$$

So this algorithm is $O(\log^2 n)$ for all $n$ and $\Theta(\log^2 n)$ for infinitely many values of $n$.

(b) To insert an element do the following:

1. Find the the smallest value $s$ such that array $A_s$ is empty. If all are full, create a new $A_{k}$ array and let $s = k$. $O(\log n)$

2. Increment the stupid $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$ array. $O(\log n)$.

3. Merge (from merge sort) array $A_0$ with $A_1$. Take the result and merge it with $A_2$. Take the result and merge it with $A_3$, etc. Finally take the result and merge it with $A_{s-1}$ and place the result in $A_s$.

Empty all arrays $A_0$ to $A_{s-1}$.

I believe it is clear that this procedure yields a structure containing the inserted element and does not violate any of the listed specifications. The run time of step 3 is on the order of the total run time for each of the $s-1$ merges performed. To merge two $m$ element arrays takes $O(m)$. Thus the total run-time of step 3 is at most a constant multiple of:

$$
\sum_{i=1}^{s} \text{length}(A_i) = \sum_{i=1}^{s} 2^i < 2^{s+1} = O(n)
$$

So the insertion takes $O(n)$. This upper bound is tight when $n = 2^m - 1$ for any integer $m$.

(c) Suppose we want to delete element $x$ and $x$ is contained in array $A_h$. Let $r$ be the smallest value such that $A_r$ is full. Take some arbitrary element from $A_r$ and place it in $A_h$ in sorted order, $O(2^h)$. Now fill arrays $A_0$ through $A_{r-1}$ with the values of $A_r$ and empty $A_r$, $O(2^r)$. Now pat yourself on the back because you’re done, $\Omega(n^m)$.

12-2 :  
There are two steps to sort the strings. First, build the radix tree consisting of the strings. Second, do a pre-order traversal of the resultant tree. The tree can be built by repeatedly entering strings into the growing tree. A string of length $l$ can be entered into a radix tree in $O(l)$. Thus, since all the lengths of the strings sum to $n$, this gives a $O(n)$ construction. Now to output the correct order, we do a pre-order traversal in which we ignore (don’t output anything) nodes that don’t correspond to actual strings. There are at most $n$ nodes in the tree, so the pre-order traversal runs in time $O(n)$.

13-2 :  
(a) After every insert or delete, traverse the tree from root to leaf along any arbitrary path. Set $bh(T)$ to be the number of black nodes you encounter on your trip. Since the height of the tree is bounded by $O(\log n)$, you won’t hurt the $O(\log n)$ running time of insert and delete.

To find the $bh$ of a given node, traverse down the tree until the node is reached. As you do this, keep track of how many black nodes you come across. Subtract this number from the $bh$ of the root to obtain the $bh$ of your target node.
(b) Traverse down the rightmost path of the tree until you reach a black node with black-height $bh[T_2]$. Such a traversal takes $O(\log n)$ operations. If such a node $y$ in the rightmost path exists, then clearly there cannot be another node with a greater key value that also has black-height $bh[T_2]$. This is because the only black nodes with larger keys would lie in $y$’s right subtree, and thus would have a larger black-height than $y$. And there must be a node $y$ in the rightmost path with black-height equal to $bh[T_2]$ because 1) $bh[T_1] \geq bh[T_2]$, 2) black-height is reduced by at most 1 as the path to the root is traversed, and 3) because the black height at the end of the path (a root) is 0.

(c) Let $T_y$ be the left subtree of $x$ and $T_2$ be the right subtree of $x$. Now place the subtree rooted at $x$ in node $y$’s old place. It is easy to verify that this creates a valid binary search tree.

(d) Make $x$ red to satisfy properties 1, 3, and 5. Run RB-INSERT-FIXUP on $x$ to satisfy properties 2 and 4.

(e) In the case that $bh[T_1] \leq bh[T_2]$ we choose our $y$ node to be the smallest keyed node in $T_2$ rather than the largest keyed node in $T_1$.

(f) As discussed we can find our $y$ in $O(\log n)$ time. We can then attach on $T_2$ and call RB-INSERT-FIXUP which takes only $O(\log n)$. And that’s all that needs to be done, I think.