1. In the previous homework, you were asked to merge $k$ vectors – $L_1, L_2, \ldots, L_k$ – whose elements are in sorted order into a single vector whose elements are again in sorted order in a sequential manner. There are of course other ways of combining the $k$ vectors. Consider the following scheme that can be thought of as a multi-dimensional Union procedure: Create a vector NewSortedVector which we will use to store all the elements of $L_1, \ldots, L_k$. Create an array $Index$. For $i = 1, \ldots, k$, $Index[i]$ will point to the earliest element in $L_i$ that has not been added to NewSorted Vector. Thus, $Index[i]$ is initially set to 0.

Next we do this step over and over again until we can’t do it anymore: among the elements at $Index[i]$ of $L_i$ for $i = 1, \ldots, k$, choose the smallest one and add it to NewSortedVector. If the element came from $L_j$, increment $Index[j]$.

Finally, return NewSortedVector.

a. Write the above procedure in pseudocode. (You’ve got to fill in details like what should be done if $Index[i]$ is “out of bounds”, and how do you keep track of the vector where the smallest element came from.) Suppose each $L_i$ has size $n$, what is the running time of the procedure? As before your running time should depend on $n$ and $k$.

Warning! If you just have two numbers, choosing the smallest number takes $O(1)$ time. If you have $k$ numbers, choosing the small number should take $O(k)$ time.

b. In the step that we repeat over and over again, notice that we’re always interested in choosing the smallest element. This suggests that a priority queue might be handy for this step. Modify your pseudocode in part (a) so that the elements at $Index[i]$ of $L_i$, $i = 1, \ldots, k$ are stored in a priority queue. Then use removeMin to extract the element with the smallest value. Again, suppose each $L_i$ has size $n$, what is the running time of this modified procedure?

2. C-2.21 in your book. Design an algorithm where you don’t have to store any values on the nodes.

3. Pretty pictures. Your book does a really good job of illustrating binary trees. Here’s why:

(i) Nodes that have the same depth lie on the same horizontal line.

(ii) For an arbitrary node $v$, nodes that are in its right subtree lie to the lower right of $v$ while those in its left subtree lie to the lower left of $v$. 
(See Figure 2.25 for example. In Figure 3.7 or 3.8, this is not exactly the case due likely to space issues.)

Let $T$ be a binary tree with $n$ nodes. Suppose you are allowed to do some traversals on $T$ and store some values on the nodes (e.g., their depth, etc.) Describe an algorithm for drawing $T$ on an $h \times n$ grid, where $h = T.\text{height}$. That is, for each node $v$ of $T$, assign it integer coordinates $(x, y)$ so that when the nodes are drawn on their corresponding coordinates, the resulting drawing has the two properties described above. (Assume that you are allowed to just say “Draw a line segment from $(x, y)$ to $(x', y')$.”) What is the running time of your algorithm?

4. Recall that we use arrays to implement heaps. Suppose the array $A[1 \ldots n]$ is storing a set of $n$ keys. Describe an algorithm that determines if $A[1 \ldots n]$ is a correct implementation of a heap. For example, if the array is $[3, 7, 15, 12, 10, 11]$ the algorithm will return “no”. What is the running time of your algorithm?