1. Assume the numbers 1 through \( n \) are stored in a binary tree \( T \). For the pre-order, postorder and inorder traversals described in class, replace the “process \( v \)” portion with “print the number stored at \( v \)”. Thus, the output of a preorder traversal of \( T \) with 5 nodes can be something like \( 2, 3, 1, 5, 4 \). We can think of this output as the “preorder traversal signature” of the tree. Clearly, we can do the same for both the postorder and inorder traversals. This problem is about these signatures.

a. Is the preorder traversal signature of a tree \( T \) unique? That is, are there two trees storing the numbers 1 through \( n \) with the same preorder traversal signature? How about the postorder traversal signature? the inorder traversal signature? If your answer is yes to any of these questions, provide an explanation. If your answer is no, come up with an example which shows two trees with the same signature.

b. Suppose you are given that the preorder traversal signature \( S_1 = [6, 2, 1, 3, 4, 5, 7, 8, 9, 10] \) and the inorder traversal signature \( S_2 = [1, 2, 4, 3, 5, 6, 7, 8, 9, 10] \) of \( T \). Reconstruct \( T \) – i.e., show the structure of \( T \) and the numbers stored in each of its nodes.

c. Generalize what you did in part (b). That is, suppose you are given \( S_1 \) and \( S_2 \), the preorder and inorder traversal signatures respectively, of \( T \). Describe an algorithm that reconstructs \( T \). You can specify \( T \) by describing each node’s left and right children.

2. Suppose \( T \) is a BinaryTree ADT. Assume, as the book does, that \( T \) is a proper binary tree. That is, if \( v \) is an internal node, it has both a left and a right child. Design an algorithm for the following operations:

\[ \text{postorderBefore}(v) \text{: returns the node visited before } v \text{ in a postorder traversal of } T. \]

\[ \text{postorderNext}(v) \text{: returns the node visited after } v \text{ in a postorder traversal of } T. \]

What is the running time of your algorithm?

Note that calling a postorder traversal on \( T \) to implement the above operations is not a legitimate answer. The goal here is for you to think about the location of the nodes that are visited before and after \( v \) in relation to \( v \) during a postorder traversal of \( T \).
3. Let $L_1$ and $L_2$ be two LIST ADT’s whose elements are distinct and sorted in increasing order. For example, each $L_i$ can contain a list of students, and the students are sorted according to their student ID number. Assume that a doubly linked list was used to implement a LIST ADT.

a. Suppose that $L_1$ and $L_2$ contain $n_1$ and $n_2$ elements respectively. Describe an $O(n_1 + n_2)$-time algorithm that combines the elements in $L_1$ and $L_2$ into one LIST ADT where the elements are still in sorted order.

b. Let us call the procedure described in part (a) UNION($L_1, L_2$) which runs in time that is linear in the size of the outcome. Suppose we now have $k$ LIST ADT’s $L_1, L_2, \ldots, L_k$ whose elements are pairwise distinct and sorted in increasing order. We again want to combine all of them into a single LIST ADT so that the elements are still in sorted order. Here’s a simple procedure for doing this: First, take the union of $L_1$ and $L_2$. Then take the union of $L_3$ and the list of $L_1 \cup L_2$. Then take the union of $L_4$ and the list of $L_1 \cup L_2 \cup L_3$, etc.

b1. Write the above procedure in pseudocode.

b2. Assuming that $L_1, \ldots, L_k$ all have $n$ elements, what is the running time of this procedure? Your answer should be in terms of $n$ and $k$.

4. This problem is a continuation of problem 1b. Here’s a “tournament” way of creating the list that combines $L_1, L_2, \ldots, L_k$. Think of it in terms of rounds. At the beginning of round $i$, suppose there are $k_i$ LIST ADT’s whose elements are pairwise distinct and sorted in increasing order: $J_1, J_2, \ldots, J_{k_i}$. For $i = 1, \ldots, \lfloor k_i/2 \rfloor$, take the union of $J_{2i-1}$ and $J_{2i}$. The resulting list moves to the next round. If $k_i$ is odd, $J_{k_i}$ simply moves to the next round. Stop when there is only one list left.

a: Write the above procedure in pseudocode.

b: Assume that $L_1, \ldots, L_k$ all have $n$ elements. What is the running time of this procedure. Again, your answer should be in terms of $n$ and $k$.

(Hint: To analyze the running time of this procedure, consider how much time is spent at each round and then add them up. Keep in mind that while the lists have size $n$ at the beginning of round 1, the lists get longer as the tournament progresses.)

c. Of the two procedures we have described, which one is more efficient? Can you provide an intuitive reason as to why this is the case?