Problem 1: We saw in class that given a binary search tree with \( n \) elements, we can output the elements in sorted order using a \( O(n) \) in-order tree walk. Use this fact and the \( \Omega(n \log n) \) lower bound on the number of comparisons needed to sort \( n \) elements, to prove a lower bound on the number of comparisons needed to build a binary search tree from a set of \( n \) elements.

Solution: Let \( C(n) \) be the number of comparisons needed to build a binary search tree with \( n \) elements.

Note that the in-order tree walk does not make any comparisons. So the following sorting algorithm that on input \( n \) elements

- build a binary search tree
- outputs the element in sorted order using an in-order tree walk

makes only \( C(n) \) comparisons. We know that any sorting algorithm must make \( \Omega(n \log n) \) comparisons, therefore \( C(n) = \Omega(n \log n) \).

Problem 2: Give an algorithm that traverses the tree and stores on each node the height of that node in the tree. Show how to maintain this information when you insert or delete nodes. Argue the correctness and analyze the complexity of your algorithm.

Solution: Note that for any node \( x \) in a tree

\[
h(x) = \text{MAX}[h(L(x)), h(R(x))] + 1 \tag{1}
\]

If you traverse the tree in a post-order walk if you visit a leaf \( z \) set \( h(z) = 0 \). When you visit an internal node \( x \) you will have visited its children already, which will allow you to set \( h(x) \) using the equation (1) above.

Remember that when you insert an element in a binary search tree, you will insert a leaf, so you set its height to 0. Then you need to ”walk back” on the path you followed to insert the new node, and update the height of the nodes in the path accordingly using equation (1) above.

Similarly, remember that when you delete a node, you always end up ”splicing” out either a leaf or a node with only one child. Again ”walk back” on the path from the node you deleted to the rooty and update the height of the nodes in the path accordingly using equation (1) above.

Problem 3: Consider a binary search tree augmented with the following information. At each node \( x \) we also store \( m(x) \): the number of nodes in the subtree rooted at \( x \) (including \( x \)). We require that for every node \( x \) in the tree \( |m(L(x)) - m(R(x))| \leq 1 \).

- Prove that \( h = O(\log n) \) where \( h \) is the height of a tree with \( n \) nodes satisfying this property [4pts]
- Show how to maintain this property when you do insertions or deletions, via rotations [6pts]

Solution:

- Intuitively this tree is fully balanced. Note that if \( m(x) \) is odd, then the two children subtrees must have the same number of nodes, while if \( m(x) \) is even the two children subtrees will differ by at most one. Intuitively a tree with \( n \) nodes under that restriction will have height \( h(n) \) such that \( h(n) \leq \log n \).

We are going to prove this by induction. Note that if \( n = 1 \) the induction hypothesis hold since \( h(1) = 0 = \log 1 \). Now we assume that \( h(m) \leq \log m \) for \( m < n \) and prove it for \( n \). Note that we are defining the height as the maximal length of a path from the root to a leaf without counting the root.

Consider now a tree with \( n \) nodes and root \( x \). If \( n = 2m + 1 \) is odd we have that \( m(L(x)) = M(R(x)) = m < n/2 \). Therefore the height of the tree is

\[
h(n) \leq h(m) + 1 \leq \log m + 1 \leq \log(n/2) + 1 = \log n - 1 + 1 = \log n
\]
If \( n = 2m \) is even we have that without loss of generality \( m(L(x)) = m \) and \( m(R(x)) = m - 1 \) with \( m = n/2 \). Then we have

\[
h(n) \leq \text{MAX}[h(m), h(m - 1)] + 1 \leq \log m + 1 = \log(n/2) + 1 = \log n
\]

- Because the tree is fully balanced it will not be possible to keep it this way with only \( O(\log n) \) operations. The condition is so strict that insertion and deletions are going to be very expensive.

After performing insertion and deletions as in a regular binary search tree you may destroy the property above. That means that you might have the situation in which for a node \( x \) (without loss of generality) \( m(L(x)) = m + 2 \) while \( m(R(x)) = m \). In this case the only solution is to find the maximum in \( L(x) \), put it in place of \( x \) and insert \( x \) in the \( R(x) \) subtree. Now \( m(L(z)) = m + 1 = m(R(z)) \). However the deletion and insertion might have messed up the subtrees, so you need to recursively fix them.

The total running time at height \( h \) is therefore \( T(h) = 2T(h - 1) + O(h) \), which is \( O(n \log n) \).