Homework7

Task 1:

Proposition: Every binary tree on n nodes where each has either zero or two children has precisely $\frac{n+1}{2}$ leaves.

- Predicate: Let P(i) be the proposition that in any binary tree consisting of i nodes, where each node either has no children or exactly two children, the tree will have exactly $\frac{n+1}{2}$ leaves.
- Base Case: P(1) is affirmed, as a binary tree with only one node will have one leaf (which is the root itself).
- Inductive Hypothesis: We assume that P(k) holds true for all k for $1 \leq k \leq n$.
- Inductive Step: Demonstrating that a binary tree with k + 2 nodes, where each node has either zero or two children, will have $\frac{k+3}{2}$ leaves.

Note: We consider k + 2 nodes due to the inherent property of binary trees with nodes having either zero or two children, resulting in an odd number of total nodes. Adding two nodes at each step maintains this oddness, hence the choice of k + 2 for the inductive step. To increase the node count from k to k + 2, we start with a tree of k nodes and select a node with zero children (a leaf), appending two children to it. This addition results in a net gain of one leaf.

After adding two nodes, the total number of leaves becomes:

$$\begin{aligned} \frac{k+1}{2} + 1 &= \frac{k+1}{2} + \frac{2}{2} \\ &= \frac{k+3}{2} \\ &= \frac{(k+2)+1}{2} \end{aligned}$$

Therefore, P(k + 2) holds true. Hence, by the principle of mathematical induction, we conclude that in any binary tree with n nodes, where each node has either zero or two children, the tree will have exactly $\frac{n+1}{2}$ leaves.

Task 2:

My code:

```
public static BinaryTreeNode buildBST(int[] keys) {
        if (keys == null || keys.length == 0) {
            return null;
        }
        Arrays.sort(keys); // Sort the keys
        return buildBST(keys, 0, keys.length - 1);
}

private static BinaryTreeNode buildBST(int[] keys, int start,
    if (start > end) {
        return null;
    }

    int mid = (start + end) / 2;
    BinaryTreeNode root = new BinaryTreeNode(keys[mid]);

    root.left = buildBST(keys, start, mid - 1);
    root.right = buildBST(keys, mid + 1, end);

    return root;
}
```

Consider each line:

- Sorting the array using the built in sort function - O(nlog(n))
- Checking if the array is null or empty - O(1)
- If statements for base cases - O(1)
- Finding the middle index - O(1)
- Creating a new Node - O(1)
- Recursive calls to buildBST - T(n/2) for each call
- Returning the root - O(1)

Total running time:
T(n) = O(nlog(n)) + O(1) + O(1) + O(1) + O(1) + 2T(n/2) + O(1)

= 2T(n/2) + O(nlog(n)) + O(1)

= 2T(n/2) + O(nlog(n))

The generated tree meets the depth requirement of (1 + log2(n) levels) as we divide the array into halves each time, reducing the range to search. After sorting the keys, we efficiently utilize the divide-and-conquer approach to construct a balanced binary search tree directly from the sorted array.

Recursively, we use the middle value as the next node's key, and repeat the steps for the left and right halves until the range collapses to one element, which serves as the base case for creating leaf nodes.

Note that by sorting the array first, we ensure that the keys are arranged in ascending order, facilitating the creation of a balanced binary search tree directly from the sorted array.

This recurrence relation solves to O(nlog(n)) since the sorting operation takes O(nlog(n)) time, and the subsequent construction of the binary search tree takes O(n) time.