hw7

Chunya Pattharapinya

6581030

1 Task 1: Proposition

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

Predicate: Let $P(i)$, for every binary tree on i nodes where each node either has no children or exactly two children. There will be $\frac{n+1}{2}$ leaves.

Base case: $P(1)$

$$\frac{1+1}{2} = 1$$

Therefore, the base case holds (1 node = 1 leaf).

Inductive Hypothesis: Assume that $P(k)$ is true. For all $1 \leq k \leq n$.

Inductive Step: WTS that it works for $n+2$ nodes.

A binary tree node contains:

1. root Node

2. two subtrees which is left node (T_L) and right node (T_R)

because each node in the binary tree has either zero or two children.

$$n+2 = 1 + n_L + n_R \textbf{ (1 is root node)}$$

Therefore:

$$n+1 = n_L + n_R$$

By IH, T_L has $\frac{n_L+1}{2}$ leaves, and T_R has $\frac{n_R+1}{2}$ leaves.

Therefore, the total leaves is:

$$\textbf{Total leaves} = \frac{n_L+1}{2} + \frac{n_R+1}{2}$$

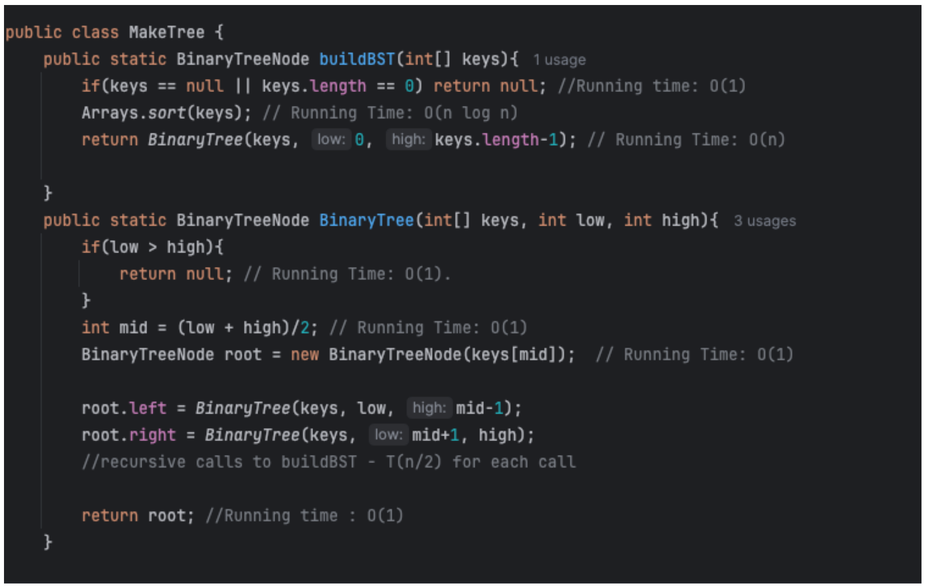
$$= \frac{(n_L+n_R)+2}{2}$$

Substitute $n_L + n_R = n+1$:

$$\frac{(n+1)+2}{2}$$

$$\textbf{Total leaves} = \frac{(n+2)+1}{2}$$

Thus, $P(n+2)$ holds true. Hence, by MI, we conclude that every binary tree with n nodes, where each node has either zero or two children, has precisely $\frac{n+1}{2}$ leaves.



```
public class MakeTree {
    public static BinaryTreeNode buildBST(int[] keys){  1 usage
        if(keys == null || keys.length == 0) return null; //Running time: O(1)
        Arrays.sort(keys); // Running Time: O(n log n)
        return BinaryTree(keys,  low: 0,  high: keys.length-1); // Running Time: O(n)

    }
    public static BinaryTreeNode BinaryTree(int[] keys, int low, int high){  3 usages
        if(low > high){
            return null; // Running Time: O(1).
        }
        int mid = (low + high)/2; // Running Time: O(1)
        BinaryTreeNode root = new BinaryTreeNode(keys[mid]);  // Running Time: O(1)

        root.left = BinaryTree(keys, low,  high: mid-1);
        root.right = BinaryTree(keys,  low: mid+1, high);
        //recursive calls to buildBST - T(n/2) for each call

        return root; //Running time : O(1)
    }
```

2 Task 2

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 algorithm constructs a binary search tree (BST) that meets the depth requirement of $1 + \log_2(n)$ levels :

1. The array is divided into halves repeatedly, with the middle element selected as the root at each step. Thus, that the tree remains balanced because the left and right subtrees have nearly equal numbers of elements.
2. Recursion continues for the left and right halves until a subarray contains only one element, forming the leaf nodes. Therefore, depth of the tree is proportional to $\log_2(n)$.
3. Sorting the array first ensures the elements are in the correct order, making it easy to build a balanced tree.

The depth of the tree is at most $1 + \log_2(n)$ because the array size is halved at each level of recursion. Sorting the array takes $O(n \log n)$, and the tree construction process takes $O(n)$. Therefore, the total time complexity of the algorithm is $O(n \log n)$.