CSE2122 (Spring 2020)
In-class activity 2, 3/6/2020

In computer science, a tree is a widely used data structure that simulates a hierarchical diagram. A tree data structure can be defined as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value and a list of references to nodes, with the constraints that no reference is duplicated, and none points to the root.

In this assignment, we will focus on binary trees. In a binary tree, each node has at most two children, which are referred to as the left child and the right child. Figure 1 shows an example of a binary tree.

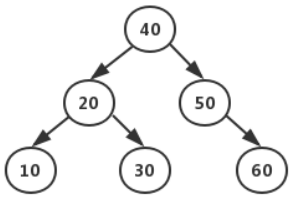

Figure 1 A Binary Tree Example

To represent binary trees, we start from the definition of a single node. Since each node consists of a value and at most two references to other nodes, we can define a binary tree node as

```
struct TreeNode{
    int data;
    TreeNode *left;
    TreeNode *right;
};
```

We will use a node's left and right pointers to point to its children nodes on either side. If a tree node has no left/right child, we set the corresponding pointer to `NULL`. Following figure shows how the binary tree in Figure 1 can be implemented with the `TreeNode` structure.

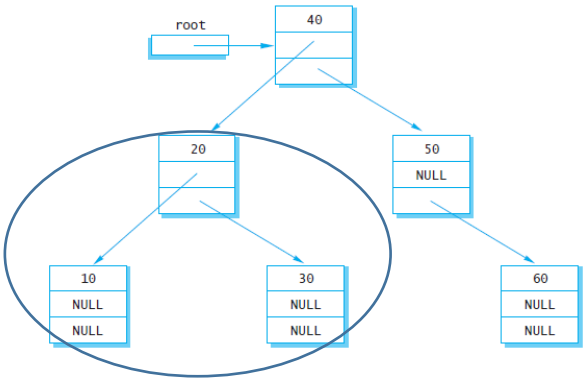

Figure 2 A Binary Tree using `TreeNode` structure (from textbook Display 13.12)

The topmost node in a tree is called the root node. The nodes that do not have child nodes are called leaf nodes.

We can use only one pointer, the root pointer, to represent a tree. The root pointer is a pointer pointing to the root node in the tree. If a tree is empty, the root pointer is set to `NULL`. For example, in figure 2, a reference to the node with value 20 can be used to represent the circled tree. Starting from the root pointer, we can easily find all nodes in a tree.

Reconsider the meaning of a node's `left` and `right` pointers. If a reference to a node can be regarded as a representation of a tree with this node as the root, we can say that a node's `left` and `right` pointers recursively point to smaller 'subtrees' on either side.

Thus, a recursive definition of the binary tree can be: a binary tree is either empty (represented by a `NULL` pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

Consider how to implement a lookup function recursively based on binary trees.

- `bool lookup(TreeNode* root, int target)`
  Given a binary search tree and a "target" value, search the tree to see if it contains the target. If yes, return `true`, otherwise return `false`.
  For example, with the binary tree in Figure 2 as input, we want `lookup(root, 40)` to return `true` and `lookup(root->left, 40)` to return `false`.

We can start from a typical case, where the input binary tree is nonempty. First we can check the value stored in the root node. If the root node has the target value, then the function can return `true`. If the root node does not has the target value, it is too early to return `false` because the target may exist in the two subtrees. We will need to check the subtrees and return whether either one of them contains the target value.

The pseudocode for the nonempty tree case can be

```
if the root value is equal to the target:
    return true
else:
    return left subtree contains the target OR right subtree contains the target
```

Questions:

1. Design an algorithm for the empty tree case
2. Give the function definition.