CS 32 Week 9 Worksheet

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.

If you have any questions or concerns, please go to any of the office hours.

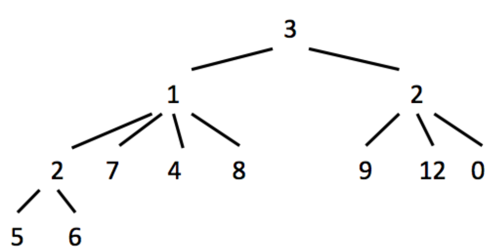
Concepts

Trees

1. Write a function that returns whether or not an integer value *n* is contained in a binary tree (that might or might not be a binary search tree). That is, it should traverse the entire tree and return true if a *Node* with the value *n* is found, and false if no such *Node* is found. (Hint: recursion is the easiest way to do this.)

```
class Node {
  int val;
  Node* left;
  Node* right;
};

bool treeContains(Node* head, int n);
```

2. Write a function that takes a pointer to a tree and counts the number of leaves in the tree. In other words, the function should return the number of nodes that do not have any children. Note that this is not a binary tree. It may help to use recursion. Example:



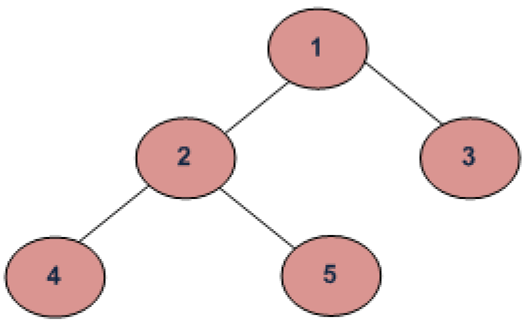
This tree has 8 leaves: 5 6 7 4 8 9 12 0

Use the following Node definition and header function to get started.

```
Node {
   int val;
   vector<Node*> children;
}

int countLeaves(Node* root);
```

3. Implement all of the following depth-first-search (DFS) traversals of a binary tree, and write the time complexity of each:

   Example Tree:



   a. Inorder Traversal: [4, 2, 5, 1, 3]
   b. Preorder Traversal: [1, 2, 4, 5, 3]
   c. Postorder Traversal: [4, 5, 2, 3, 1]

```
struct Node {
     int val;
     Node* left, right;
};

vector<int> inorderTraversal(Node* root) {
     // Fill in code
}
vector<int> preorderTraversal(Node* root) {
     // Fill in code
}
vector<int> postorderTraversal(Node* root) {
     // Fill in code
}
```

4. Given an in-order and post-order traversal of the same tree, draw the binary tree.

   in-order: 4, 2, 5, 1, 6, 7, 3, 8

post-order: 4, 5, 2, 6, 7, 8, 3, 1

Draw the tree.

5. Write the following recursive function:

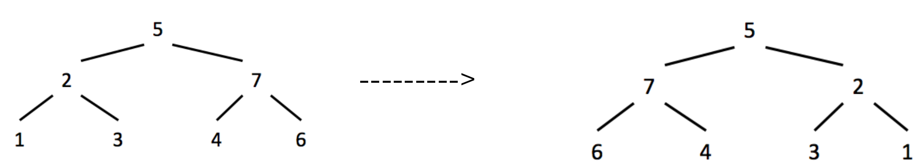
```
bool isSubtree(Node* main, Node* potential);

struct Node {
      int val;
      Node* left, right;
};
```

The function should return true if the binary tree with the root, `potential`, is a subtree of the tree with root, `main`. You may use any helper functions necessary. A subtree of a tree `main` is a tree `potential` that contains a node in `main` and all of its descendants in `main`.

```
  10  ← potential                      26              ←            main
 / \                                  /                               \
4   6                               10                                 3
                                   /          \                         \
                                 4    6    17
```

In this example, `isSubtree(main, potential)` would return true.

6. Write a function that takes a pointer to the root of a binary tree and recursively reverses the tree. Example:



Use the following Node definition and header function to get started.

```
Node {
   int val;
   Node* left;
   Node* right;
};
```

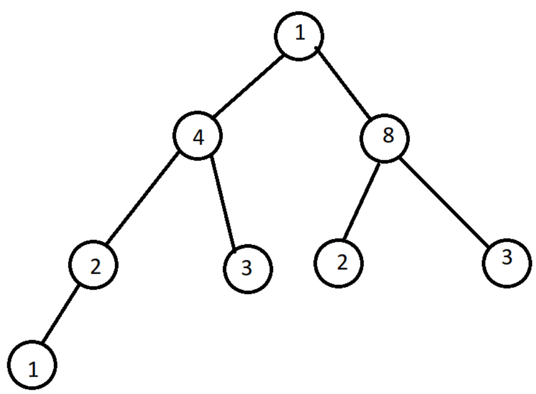
```
void reverse(Node* root);
```

7. Implement the following function, `getGreatestPath`.

```
int getGreatestPath(Node* head);

struct Node {
      int val;
      Node* left, right;
};
```

The value of a path in a binary tree is defined as the sum of all the values of the nodes within that path. This function takes a pointer to the head of a binary tree, and it finds the value of the path from the head to a leaf with the greatest value.

Ex: The following tree has a greatest path value of 12 (1->8->3).



8. Given a binary tree, write a function that returns all root-to-leaf paths. What is the time complexity of this function? For example, given the following binary tree:

```
  1
 / \
2   3
 \
  5
```

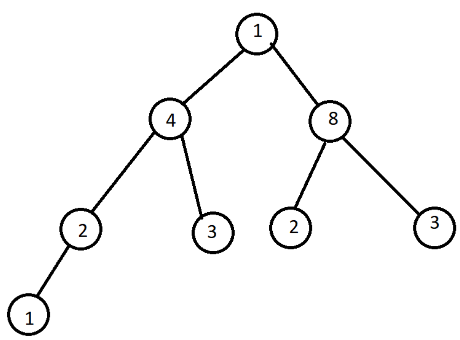
All root-to-leaf paths are:
["1->2->5", "1->3"]

```
struct Node {
      int val;
```

```
	Node* left, right;
};

vector<std::string> rootToLeaf(Node* head) {
	// Fill in code
}
```

9. Write a function that finds the maximum depth of a binary tree. A tree with only one node has a depth of 1; let's decree that an empty tree has a depth of -1.

```
struct Node {
	int val;
	Node* left, right;
};

int maxDepth(Node *root);
```



This tree has a maximum depth of 3.

10. Write a function that does a level-order traversal of a binary tree and prints out the nodes at each level with a new line between each level.

```
        1   <-- root
       /  \
      2    3
     / \    \
    4   5    6
   / \  /
  7   8 9
```

Function declaration: `void levelOrder(Node* root).`

If the root from the tree above was passed as the parameter, levelOrder should print:

1
2 3
4 5 6
7 8 9

Then analyze the time complexity of your algorithm.

11. Implement a level-order traversal of a tree with two queues this time (if you haven't already done so). Analyze the time complexity.
    Hint: We can insert the first level in first queue and print it, and while popping from the first queue, insert its left and right nodes into the second queue. Now, start printing the second queue and before popping, insert its left and right child nodes into the first queue. Continue this process till both the queues become empty.

12. Write two functions. The first function, tree2Vec, turns a binary tree of positive integers into an vector. The second function, vec2Tree, turns a vector of integers into a binary tree. The key to these functions is that the first function must transform the tree such that the second function reverses it. So,

    ```
    tree2Vec(vec2Tree(vec)) == vec.
    ```

    ```
    Node* vec2Tree(vector<int> v);
    vector<int> tree2Vec(Node* root);
    ```

    *Hint: Think about how to encode the parent-child relationship of a tree inside of a vector.
    *Hint 2: Note the vector length doesn't need to equal the number of nodes