CS 32 Worksheet Week 8

Concepts: Sorting, Trees

Q1. (5 min) Consider the following array that has just been partitioned by the first step of the quicksort algorithm:

3 0 2 4 5 8 7 6 9

Could 5 be the initial pivot? What about 7? Explain why or why not.

Q2. (5 min) Given the following array of integers, write down what the vector will look like for 3 iterations of a quicksort algorithm with the last element being the pivot.

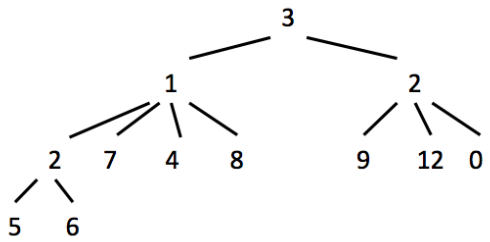
25 56 18 20 12 9 15

Q3. (8 min) 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.)

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

Q4. (8 min) 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:



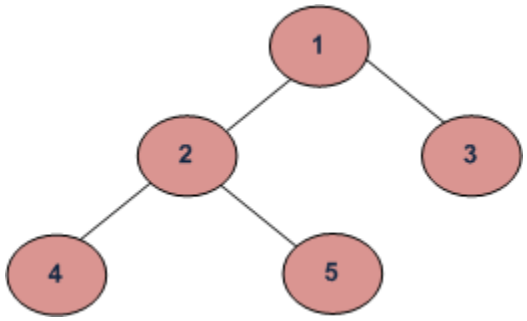
This tree has 8 leaves: 5 6 7 4 8 9 12 0

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

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

Q5. (15 min) Implement all of the following depth-first-search (DFS) traversals of a binary tree, give each traversal result, and write the time complexity of each:

Example Tree:




```
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
}
```

Show your implementation:

Traversal result:
   Inorder Traversal:
   Preorder Traversal:
   Postorder Traversal:
Time complexity:
   Inorder Traversal:
   Preorder Traversal:
   Postorder Traversal:

Q6. (10 min) 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`.

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

Q7. (5 min) 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.

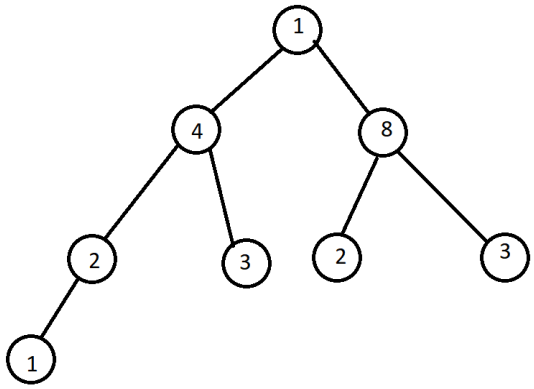
```
Node {
   int val;
   Node* left;
   Node* right;
};
```

Q8. (5 min) Implement the following function, `getGreatestPath`.

```
int getGreatestPath(Node* head);

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

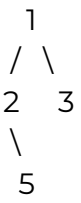
The value of a path in a binary tree is defined as the sum of all the values of the nodes within that path. This function takes a pointer to the head of a binary tree, and it finds the value of the path from the head to a leaf with the greatest value.

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




```
int getGreatestPath(Node* head) {



}
```

Q9. (8-10 min) 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:



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
}
```

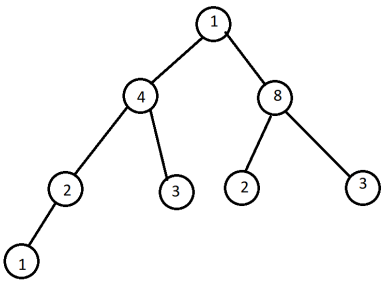
Additional Practice Week 8

Q1. 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

Q2. Write a function that finds the maximum depth of a binary tree. A tree with only one node has a depth of 0; let's decree that an empty tree has a depth of -1.

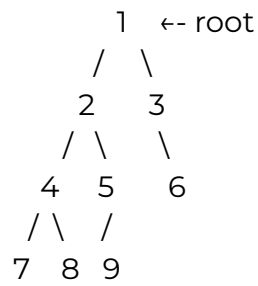
```
struct Node {
      int val;
      Node* left, right;
};

int maxDepth(Node *root);
```



This tree has a **maximum depth of 3**.

```
int maxDepth(Node *root) {









}
```

Q3. 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.



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.

Time Complexity =

Q4. 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 the 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.

Time Complexity =

Q5. Write two functions. The first function, tree2Vec, turns a binary tree of positive integers into a 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 1: 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