CS 32 Worksheet 5

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 contact mcmallett@ucla.edu or sarahelizabethcastillo@gmail.com, or go to any of the LA office hours.

Concepts

Algorithm Analysis, Trees

1. Consider this function that returns whether or not an integer is a prime number:

```
bool isPrime(int n) {
  if (n < 2 || (n % 2 == 0 && n != 2)) return false;

  for (int i = 3; (i * i) <= n; i += 2) {
    if (n % i == 0) return false;
  }
  return true;
}
```

   What is its time complexity?

   Time: 2 min (easy)

2. 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.)

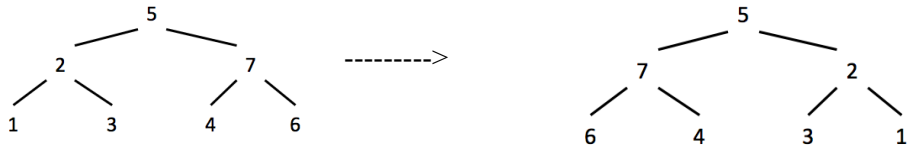
```
struct Node {
  int val;
  Node* left;
  Node* right;
};

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

   Time: 15 min (medium)

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

   Example:



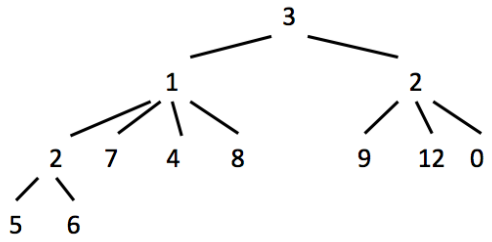
   Use the following Node definition and header function to get started.

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

void reverse(Node* root);
```

   Time: 6 min (medium)

4. 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. Also, it would probably help to use recursion. Example:



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

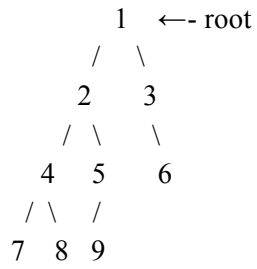
   Use the following Node definition and header function to get started.

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

int countLeaves(const Node* root);
```

   Time: 15 min (medium)

5. 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 printLevelOrder(const Node* root).`
If the root from the tree above was passed as the parameter, printLevelOrder should print:

1
2 3
4 5 6
7 8 9

Then analyze the time complexity of your algorithm for the worst case.

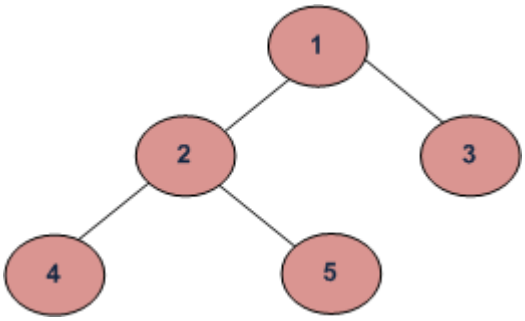
Time: 7-15 min. (medium-hard)

6. Implement a level-order traversal of a tree with a queue this time (if you haven't already done so). Analyze the time complexity.
Hint: Process a level starting when the queue contains all and only the nodes at that level; note how many there are in the queue then. One at a time, take that many nodes from the queue and add their children to the queue.

Time: 30-40 min. (hard)

7. 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
}
```

Time: 5-10 min. (medium)

8. Given a binary tree, return 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
}
```

Time: 30-40 min. (medium-hard)

9. What is the time complexity of the following code? **(Time: 2-5 min; medium)**

```
int randomSum(int n) {
```

```
  int sum = 0;
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < i; j++) {
      if(rand() % 2 == 1) {
        sum += 1;
      }
      for(int k = 0; k < j*i; k+=j) {
        if(rand() % 2 == 2) {
          sum += 1;
        }
      }
    }
  }
  return sum;
}
```

Time: 2-5 min. (medium)

10. Write two functions. The first function, tree2Vec, turns a binary tree of 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 can 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

    Time: 45 min. (hard)

11. Find the time complexity of the following function

    ```
    int obfuscate(int a, int b) {
          vector<int> v;
          set<int> s;
          for (int i = 0; i < a; i++) {
                v.push_back(i);
                s.insert(i);
          }
          v.clear();

          int total = 0;
    ```

```
      if (!s.empty()) {
            for (int x = a; x < b; x++) {
                  for (int y = b; y > 0; y--) {
                        total += (x + y);
                  }
            }
      }
      return v.size() + s.size() + total;
}
```

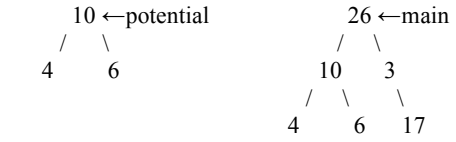
Time: 2-5 min. (medium)

12. Write the following recursive function:

```
bool isSubtree(const Node* main, const 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`.



`isSubtree(main, potential)` would return true.

Time: 45 min. (hard)

13. Implement the following function, `greatestPathValue`.

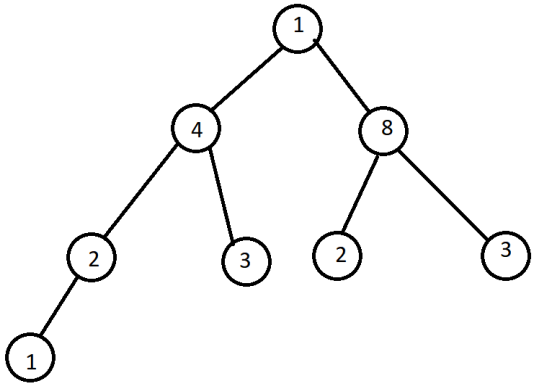
```
int greatestPathValue(const Node* head);

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

The value of a path in a binary tree is defined as the sum of the values of all the nodes in that

path. This function takes a pointer to the root of a binary tree, and it finds the value of the path from the root to a leaf with the greatest value.

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



Time: 15 min. (medium)