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. Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

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?

   O($\sqrt{n}$). The function's loop only runs while i*i <= n, or i <= $\sqrt{n}$. Although i is incremented by 2 each time (i += 2), this would be ½$\sqrt{n}$ iterations and the coefficient ½ can be dropped in big O analysis. Note that $\sqrt{n}$ is $n^{½}$, and exponents of n do matter.

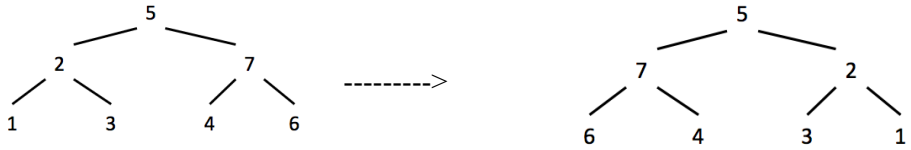
2. Write a function that returns whether or not an integer value *n* is contained in a binary tree (that might or might not be a binary search tree). That is, it should traverse the entire tree and return true if a *Node* with the value *n* is found, and false if no such *Node* is found. (Hint: recursion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

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

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

bool treeContains(const Node* head, int n) {
  if (head == nullptr) {
    return false;
  } else if (head->val == n) {
    return true;
  } else {
    return treeContains(head->left, n) ||
           treeContains(head->right, n);
  }
}
```

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

Example:

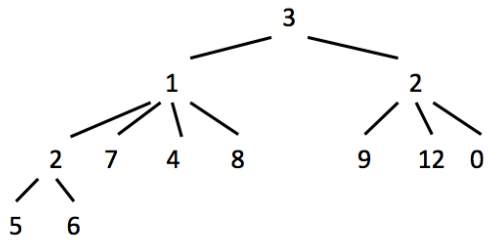


Use the following Node definition and header function to get started.

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

void reverse(Node* root) {
    if (root != nullptr) {
        Node* temp = root->left;
        root->left = root->right;
        root->right = temp;

        reverse(root->left);
        reverse(root->right);
    }
}
```

Notice that this approach has a preorder traversal structure: deal with the node, then deal with its subtrees. Would the algorithm work if it were postorder (i.e., do the two recursive calls first, then swap the pointers in the node itself)?

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:

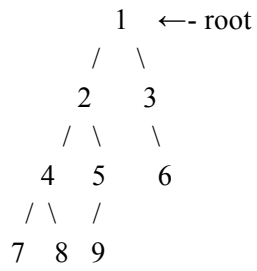


This tree has 8 leaves: 5 6 7 4 8 9 12 0

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

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

int countLeaves(const Node* root) {
      if (root == nullptr) // no leaves from this node
            return 0;
      if (root->children.size() == 0) // this node is a leaf
            return 1;

      int count = 0;
      for (int i = 0; i < root->children.size(); i++)
      {
            count += countLeaves(root->children[i]);
      }
      return count;
}
```

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(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.

Note that this is just one solution. Look at the next problem for another possible approach.

```
/* Function to print level order traversal a tree*/
void printLevelOrder(const Node* root)
{
    int h = height(root);
    for (int i = 1; i <= h; i++)
    {
        printGivenLevel(root, i);
        cout << endl;
    }
}

/* Print nodes at a given level */
void printGivenLevel(const Node* root, int level)
{
    if (root == nullptr)
        return;
    if (level == 1)
        cout << root->data << " ";
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}

/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(const Node* root)
{
    if (root == nullptr)
        return 0;
    else
```

```
    {
        /* compute the height of each subtree */
        int lheight = height(root->left);
        int rheight = height(root->right);

        /* use the larger one */
        if (lheight > rheight)
            return lheight+1;
        else
            return rheight+1;
    }
}
```

Note first that the height function has to visit every node, so it's O(n).

Assume n is the number of nodes in the tree. The algorithm works by visiting all (i.e., the one) nodes at level 1, then all the nodes at levels 1 and 2, then all the nodes at levels 1, 2, and 3, etc. (It only prints the last level it reaches each time, but it has to pass through all higher levels to get there.) The greater the height of the tree, the more times nodes are visited over and over. In the worst case, the tree is totally skewed so that each node has one empty subtree and one with all the rest of the nodes -- a linear path from top to bottom. printGivenLevel visits every node down to *level*, which in this worst case is *level* nodes. printLevelOrder calls printGivenLevel for each value of *level* down to the bottom, which in this worst case in n, so it visits $1 + 2 + 3 + ... + n = n(n-1)/2 = O(n^2)$ nodes. This dominates the height function's O(n), so the whole answer is $O(n^2)$ in the worst case.

(In the best cases, the height of the tree is minimal so that nodes are revisited as few times as possible. The levels above the leaves should be as full as possible. This makes the height (log n). The analysis is beyond the scope of this class, involving evaluating sum(k=0 to (log n)-1) of $2^k((\log n)-k))$, but the result would be O(n log n).)

6. 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.

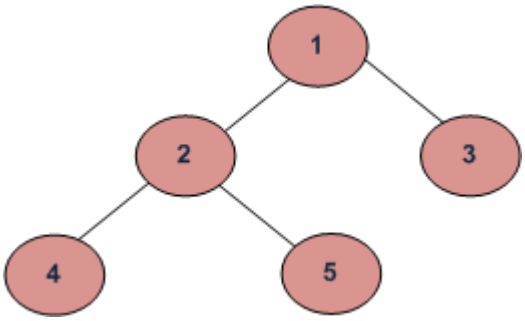
```
#include <iostream>
#include <queue>
using namespace std;

// Iterative method to do level order traversal line by line
void printLevelOrder(const Node *root)
{
```

```
    // Special case
    if (root == nullptr) return;

    // Create an empty queue for level order traversal
    queue<const Node*> q;

    // Enqueue root
    q.push(root);

    while ( ! q.empty())
    {
        // Dequeue all nodes of current level and Enqueue all
        // nodes of next level
        for (int nodeCount = q.size(); nodeCount>0; nodeCount--)
        {
            const Node* node = q.front();
            q.pop();
            cout << node->data << " ";
            if (node->left != nullptr)
                q.push(node->left);
            if (node->right != nullptr)
                q.push(node->right);
        }
        cout << endl;
    }
}
```

Each of the n nodes is pushed/popped/printed once; the time complexity of this algorithm is O(n).

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;
};

// We can approach these problems both recursively and
// iteratively. We will use the recursive solutions here,
// for which we will need helper functions.

// Helper void function with reference to vector
void inorder(Node* root, vector<int>& nodes) {
      if (root == nullptr) return;
      inorder(root->left, nodes);
      nodes.push_back(root->val);
      inorder(root->right, nodes);
}
// Use helper
vector<int> inorderTraversal(Node* root) {
      vector<int> nodes;
      inorder(root, nodes);
      return nodes;
}
// Helper void function with reference to vector
void preorder(Node* root, vector<int>& nodes) {
      if(root == nullptr) return;
      nodes.push_back(root->val);
      preorder(root->left, nodes);
      preorder(root->right, nodes);
}
// Use helper
vector<int> preorderTraversal(Node* root) {
      vector<int> nodes;
      preorder(root, nodes);
      return nodes;
}
// Helper void function with reference to vector
void postorder(Node* root, vector<int>& nodes) {
      if(root == nullptr) return;
      postorder(root->left, nodes);
      postorder(root->right, nodes);
      nodes.push_back(root->val);
}
// Use helper
vector<int> postorderTraversal(Node* root) {
```

```
        vector<int> nodes;
        postorder(root, nodes);
        return nodes;
}
```

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;
};

// We will use a helper function to keep a track of the current
// path

void rootToLeaf(Node* root, vector<std::string>& paths,
                                                std::string curr) {
        if (root->left == nullptr  &&  root->right == nullptr) {
                paths.add(curr);
                return;
        }
        if (root->left != nullptr) {
                rootToLeaf(root->left, paths, curr + "->" +
                                        std::to_string(root->left->val));
        }
        if (root->right != nullptr) {
                rootToLeaf(root->right, paths, curr + "->" +
                                        std::to_string(root->right->val));
        }
}
// Use helper
vector<std::string> rootToLeaf(Node* root) {
        vector<std::string> paths;
        if (root != nullptr)
                rootToLeaf(root, paths, std::to_string(root->val));
        return paths;
```

```
}
```

9. What is the time complexity of the following code?

```
int randomSum(int n) {
  int sum = 0;
  for(int i = 0; i < n; i++) { // this loop runs n times
    for(int j = 0; j < i; j++) { // this loop runs i (which goes
up to n) times for each iteration of i loop
      if(rand() % 2 == 1) {
        sum += 1;
      }
      for(int k = 0; k < j*i; k += j) {
// this loop runs (j*i / j) times, or i (which goes up to n)
times, for each iteration of j loop
        if(rand() % 2 == 2) {
          sum += 1;
        }
      }
    }
  }
  return sum;
}
```

```
O(n^3), note it doesn't matter that "if(rand() %2 == 2)" is
always false, rand is still run each time
```

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

```
bool hasNonNull(vector<Node*>& v) {
  for (int i = 0; i < v.size(); i++) {
    if (v[i] != nullptr) {
      return true;
    }
```

```
  }
  return false;
}

vector<int> tree2Vec(Node* root) {
  vector<int> myV;
  vector<Node*> curLevel;
  vector<Node*> nextLevel;
  curLevel.push_back(root);
  myV.push_back(root->value);
  while(hasNonNull(curLevel)) {
    // cout << "New Iteration" << endl;
    // printVector(curLevel);
    for(int i = 0; i < curLevel.size(); i++) {
      // printVector(nextLevel);
      Node* curNode = curLevel[i];
      if(curNode == nullptr) {
        myV.push_back(-1);
        myV.push_back(-1);
        nextLevel.push_back(nullptr);
        nextLevel.push_back(nullptr);
        continue;
      }

      nextLevel.push_back(curNode->left);
      if(curNode->left == nullptr){
        myV.push_back(-1);
      } else {
        myV.push_back(curNode->left->value);
      }

      nextLevel.push_back(curNode->right);
      if(curNode->right == nullptr) {
        myV.push_back(-1);
      } else {
        myV.push_back(curNode->right->value);
      }
    }

    curLevel = nextLevel;
    nextLevel.clear();
  }

  return myV;
}
```

```
Node* vec2TreeHelper(vector<int> v, int i, Node* nodeI) {
  int leftIndex = i*2+1;
  int rightIndex = i*2+2;
  int leftValue = v[leftIndex];
  int rightValue = v[rightIndex];

  if(leftValue != -1) {
    nodeI->left = new Node;
    nodeI->left->value = leftValue;
    vec2TreeHelper(v, leftIndex, nodeI->left);
  } else {
    nodeI->left = nullptr;
  }

  if(rightValue != -1) {
    nodeI->right = new Node;
    nodeI->right->value = rightValue;
    vec2TreeHelper(v, rightIndex, nodeI->right);
  } else {
    nodeI->right = nullptr;
  }

  return nodeI;
}

Node* vec2Tree(vector<int> v) {
  Node* n = new Node;
  n->value = v[0];
  // n->left = nullptr;
  // n->right = nullptr;
  return vec2TreeHelper(v, 0, n);
}
```

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++) { // loop runs a times
            v.push_back(i);
            s.insert(i);  // set uses binary search tree for
insert, which is logn, or in this case, log a for each insert
      }
      v.clear(); // O(a), linear time to delete every element
```

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

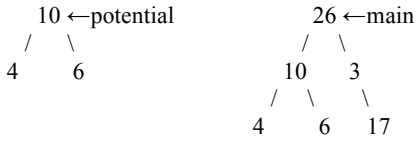
The first for loop takes a log a, which means the one v.clear() O(a) step can be ignored. The second for loop is O(b^2). Note that while n^2 > n log n, you cannot drop a log a because a is a separate input from b.

Overall time complexity: O(a log a + b^2)

12. 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`.



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

```
// helper function
bool identical(const Node* r1, const Node* r2)
{
      // Base case
      if (r1 == nullptr && r2 == nullptr)
            return true;
```

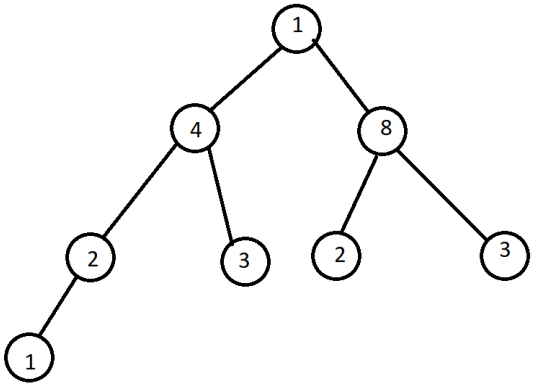
```
        if (r1 == nullptr || r2 == nullptr)
                return false;

        // Check if the data of both roots is same
        // and data of left and right subtrees are also same
        return (r1->val == r2->val && identical(r1->left, r2->left)
&& identical(r1->right, r2->right) );
}

bool isSubtree(const Node* main, const Node* potential)
{
        // Base cases
        if (potential == nullptr)
                return true;

        if (main == nullptr)
                return false;

        if (identical(main, potential))
                return true;

        return      isSubtree(main->left, potential) ||
                    isSubtree(main->right, potential);
}
```

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).




```
int greatestPathValue(const Node* head) {
      if (head == nullptr)
        return 0;
      int leftMax = greatestPathValue(head->left);
      int rightMax = greatestPathValue(head->right);

      if (leftMax > rightMax)
        return head->val + leftMax;
      else
        return head->val + rightMax;
}
```