CS 32 Solutions 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.

Solution

5 could be the initial pivot since all elements to the left of it are less than 5 and all elements to the right of it are greater than 5.
7 could not be the initial pivot since there are elements greater than 7 to the
left of it and elements less than 7 to the right of i

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

Solution

The pivots that resulted in the listed iterations are underlined.

1st iteration -> 12 9 **15** 25 56 18 20
(note that 12, 9 may appear in a different order, as may 25, 56, 18, 20)

2nd iteration -> 9 12 15 18 **20** 25 56
(note that 25, 56 may appear in a different order)

3rd iteration -> 9 12 15 18 20 25 **56**

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.)

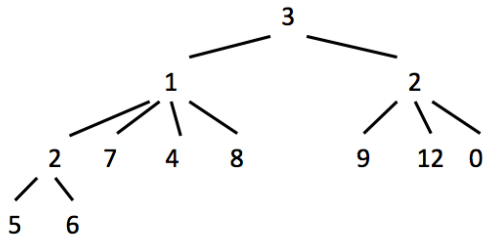
```
struct Node {
  int val;
  Node* left;
  Node* right;
};

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

Solution

```
bool treeContains(Node* head, int n) {
  // Base case
  if (head == nullptr) {
    return false;
  } else if (head->val == n) {
    return true;
  } else {
    // Check all children
    return treeContains(head->left, n) ||
           treeContains(head->right, n);
  }
}
```

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.

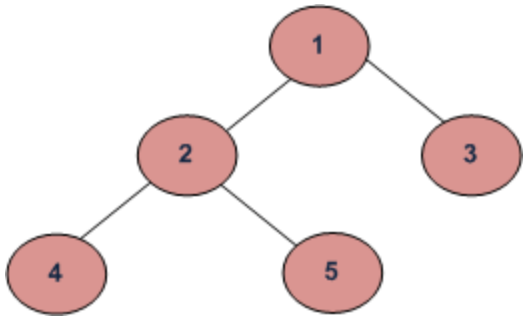
```
Node {
   int val;
   vector<Node*> children;
}

int countLeaves(Node* root);
```

Solution

```
int countLeaves(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;
}
```

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
}
```

Solution

```
// 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);
```

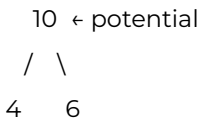
```
        return nodes;
}
// Helper void function with reference to vector
void preorder(Node* root, vector<int>& nodes) {
        if(root == nullptr) return;
        nodes.push_back(root->val);
        preorder(root->left, nodes);
        preorder(root->right, nodes);
}
// Use helper
vector<int> preorderTraversal(Node*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

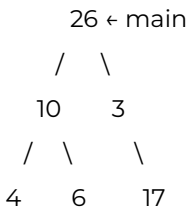
```
Traversal result:
   Inorder Traversal: [4, 2, 5, 1, 3]
   Preorder Traversal: [1, 2, 4, 5, 3]
   Postorder Traversal: [4, 5, 2, 3, 1]
Time complexity:
   Inorder Traversal: O(n)
   Preorder Traversal: O(n)
   Postorder Traversal: O(n)
```

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`.





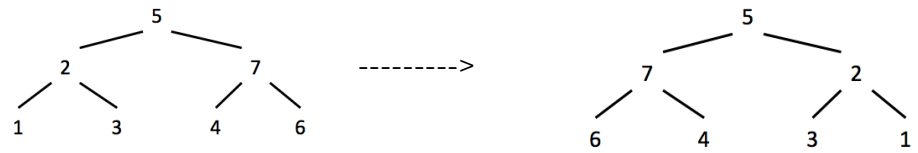
In this example, `isSubtree(main, potential)` would return true.

Solution

```
      // helper function
bool identical(Node* r1, 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(Node* main, 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);
}
```

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;
};

void reverse(Node* root);
```

Solution

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

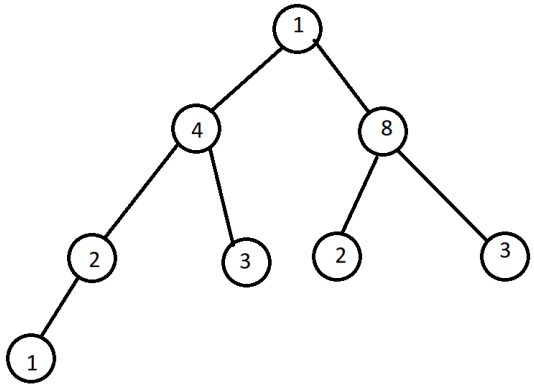
```
    root->left = root->right;
    root->right = temp;

    reverse(root->left);
    reverse(root->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).+



Solution

```
int getGreatestPath(Node* head) {
  if (head == nullptr)
    return 0;
```

```
  int leftMax = getGreatestPath(head->left);
  int rightMax = getGreatestPath(head->right);

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

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:

```
    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
}
```

Solution

```
// 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.push_back(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;
}
```

Time complexity is O(n).

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

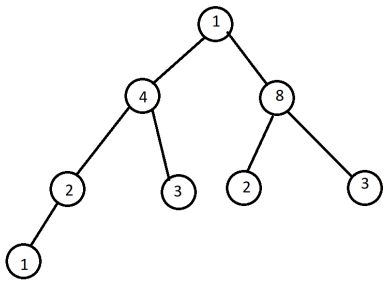
Solution

```
                    1
                  /   \
                 /     \
                2       3
               / \     / \
              4   5   7   8
                     /
                    6
```

Problem, solution, and explanation from https://www.programcreek.com/2013/01/construct-binary-tree-from-inorder-and-postorder-traversal/

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);
```



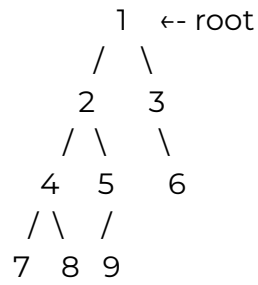
This tree has a **maximum depth of 3**.

Solution

```
int maxDepth(Node* root) {
    if (root == nullptr)
```

```
        return -1;
    // computer depth of each subtree
    int lDepth = maxDepth(root->left);
    int rDepth = maxDepth(root->right);

    // return the max of the two
    return max(lDepth, rDepth)+1;
}
```

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.

Solution

```
/* Function to print level order traversal a tree*/
void printLevelOrder(Node* root) {
    int h = height(root);
    int i;
```

```
    for (int i = 1; i <= h; i++) {
        printGivenLevel(root, i);
         cout << endl;
    }
}


/* Print nodes at a given level */
/* It visits no more than 2^(level-1) nodes of the tree */
void printGivenLevel(Node* t, int level) {
    if (root == nullptr)
        return;
    if (level == 1)
        cout << t->data << " ";
    else if (level > 1)
    {
        printGivenLevel(t->left, level-1);
        printGivenLevel(t->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(Node* t) {
    if (node == nullptr)
        return 0;

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

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

```
            return rheight+1;
    }
}
```

In the worst case, we would end up with a tree height of n, making our Time Complexity: Height O(n), 1+2+3+...+n=n(n+1)/2, O(n^2)

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.

Solution

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

// Iterative method to do level order traversal line by line
void printLevelOrder(Node *root) {
    // Base Case
    if (root == nullptr)  return;

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

    // Enqueue Root and initialize height
    q.push(root);

    while ( ! q.empty())
    {
        // Dequeue all nodes of current level and Enqueue all
        // nodes of next level
        int nodeCount = q.size();
        while (nodeCount > 0)
```

```
        {
            Node* node = q.front();
            cout << node->data << " ";
            q.pop();
            if (node->left != nullptr)
                q.push(node->left);
            if (node->right != nullptr)
                q.push(node->right);
            nodeCount--;
        }
        cout << endl;
    }
}
```

The time complexity of this algorithm is O(n).

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

Solution

```
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)) {
    for(int i = 0; i < curLevel.size(); i++) {
      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);
}
```