SIG Algorithm Challenges

Week 4: Trees

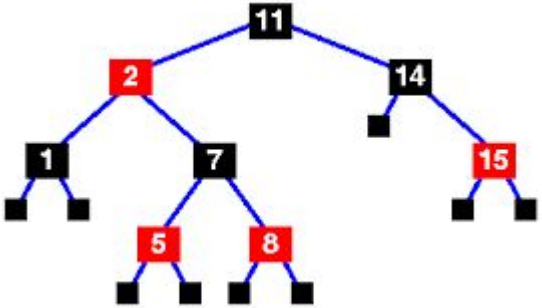
Dane's Contact Email: dziema2@uic.edu

Join our Slack channel, #sig_algorithm_chal on the UIC ACM slack

Tree Basics

Trees are a data structures consisting of nodes containing some value connected to other nodes with no cycles

- Most common trees are binary trees (2 children per node)
  - Can be n-ary (n children per node)
- Most common trees are rooted (have an explicit root node with unidirectional connections away from root)
  - Unrooted trees are generally considered acyclic graphs
- Binary search trees are a common type of tree
  - All left children are less than their root, all right children are greater
- Trees generally have runtimes proportional to their height, so balanced trees are preferred
  - AVL trees and Red-Black trees are examples of height balancing trees
  - Red-Black trees are used more commonly due to fewer adjustments on average per insertion
- All subtrees of a tree are trees themselves
  - Makes recursion very useful for trees



Identifying Tree Problems

Most explicit tree problems (i.e. ones where you need to use some TreeNode class) are obvious.

- There are some implicit tree problems that are a bit harder to recognize, see “kthGrammar” and “differentParenthesis” from week 2 for examples of these.

- These sorts of problems can be solved without trees, but having a knowledge of trees helps

- Recursion and trees are often closely related, if given a recursion problem often times thinking of the subproblem structure as a tree helps. Likewise most tree problems can be solved recursively.

General Tips for Tree problems

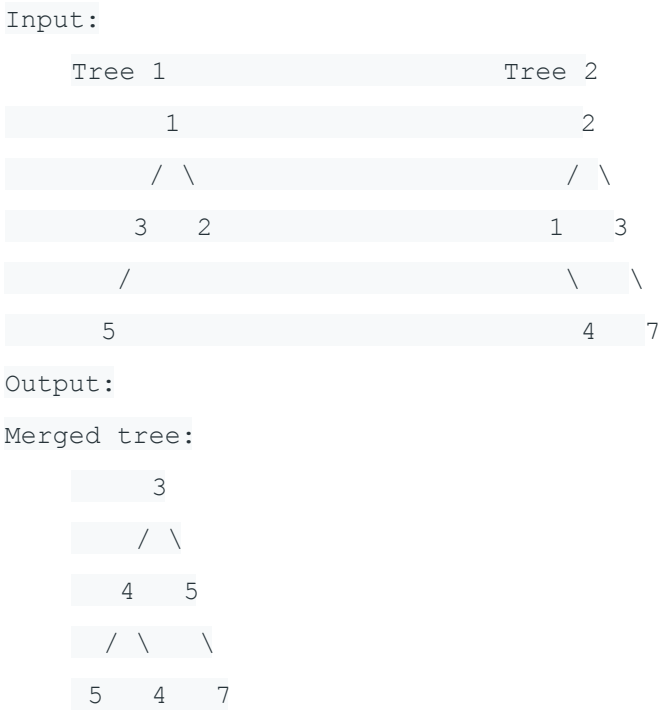
- **RECURSION**
  - Tree problems don't have to be solved recursively, but normally the iterative version is 50-100 lines while the recursive version is ~10 lines
  - All subtrees are themselves trees
  - Generally tree algorithms can be built by thinking "If I had some function that could solve this problem for my left and right subtree, how could I solve it for the tree rooted at this node"
  - Use recursion and your base cases as this "some function"

- Don't assume all trees are BSTs
  - Trees don't have to be binary, or even rooted
  - If not specified, a tree is most likely not a BST, can't assume left child is less and right is greater

- Know your run times
  - Run times are generally O(H)
  - If a tree is not balanced, run time is likely O(n), as O(log(n)) only occurs for balanced trees
  - Even with balanced trees, sometimes you have to search the whole tree and get O(n)

Sample Tree Problem

Given two binary trees, merge them together. If two nodes overlap, add the two values together



Problem Approach

Use recursion to solve this. Base case: a tree merged with an empty tree is unchanged

1: Use recursion to recursively merge the left and right subtrees of each tree

2: After the right and left are merged, merge the two root nodes

Pseudocode for this problem

```
MergeTrees(rootA, rootB):
        If rootA == null
                return rootB
        Else if rootB == null
                return rootA
        Else
                Create a new tree T  to hold the merged tree
                T.left = MergeTrees(rootA.left, rootB.left).
                T.right = MergeTrees(rootA.right, rootB.right);
                Set the roots value equal to the sum of the two root values
                Return T
```

Java Code for this problem

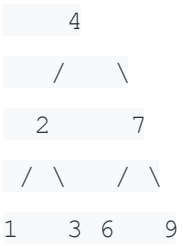
```
class Solution {
    public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
        //if one of the trees is null then just return the other
        if (t1 == null) {
            return t2;
        }
        else if (t2 == null) {
            return t1;
        }
        //otherwise add the two values at the root together and merge
        //the right and left subtrees recursively
        else {
            TreeNode T = new TreeNode(t1.val + t2.val);
            T.left = mergeTrees(t1.left, t2.left);
            T.right = mergeTrees(t1.right, t2.right);
            return T;
        }
    }
}
```

Sample Tree Problem

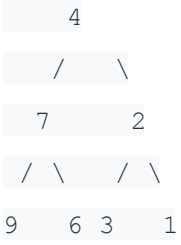
Invert a binary tree.

Example:

Input:



Output:



Problem Approach

Use recursion to solve this. Base case: an empty tree reversed is unchanged

1: Use recursion to recursively invert the left and right subtrees of each tree

Pseudocode for this problem

```
InvertTrees(root):

    If root == null

        Return root

    Else

        Root.right = InvertTrees(root.left)

        Root.left = InvertTrees(root.right)
```

Java Code for this problem

```
class Solution {
    public TreeNode invertTree(TreeNode root) {
        //empty trees are already inverted
        if (root == null) {
            return root;
        }
        //otherwise invert the right and the left
        //then swap the left and right subtree at the root
        else {
            TreeNode temp = root.right;
            root.right = invertTree(root.left);
            root.left = invertTree(temp);
            return root;
        }
    }
}
```

Tree Problems to Work On

Links to level 1-4 can be found at https://github.com/dane8373/SIG_Algorithm_Challenges/tree/master/Week4

Level 1) Given a binary tree, find its maximum depth.

Level 2) Given a binary search tree and the lowest and highest boundaries as L and R, trim the tree so that all its elements lies in [L, R] (R >= L).

Level 3) Given an n-ary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

Level 3b) Given a binary tree, determine if it is a valid binary search tree (BST).

Level 4) Given the root of a binary tree with N nodes, each node in the tree has node.val coins, and there are N coins total.

In one move, we may choose two adjacent nodes and move one coin from one node to another. (The move may be from parent to child, or from child to parent.) Return the number of moves required to make every node have exactly one coin.

Problem Approach for Level 1 problem

Surprise: We are using recursion for this

1: Null trees have height 0

2: If a tree is not null, it has height of the max of their left and right subtree heights +1

Java Code for level 1 problem

```java
class Solution {
    public int maxDepth(TreeNode root) {
        //null trees have 0 depth
        if (root == null) {
            return 0;
        }
        int left = maxDepth(root.left);
        int right = maxDepth(root.right);
        //return the max of the left and right + 1
        return left > right ? 1+left : 1+right;
    }
}
```

Problem Approach for Level 2 problem

More recursion!

1: If a tree is null then it doesn’t need to be pruned at all

2: If a node is not null and it is greater than the max, then this node and its whole right subtree must be removed. Same idea if the node is less than the min

3: Otherwise, leave this node alone and prune the left and right subtrees

Java Code for level 2 problem

```
class Solution {
    public TreeNode trimBST(TreeNode root, int L, int R) {
        //null subtrees are already trimmed
        if (root == null) {
            return root;
        }
        //if this value is less than the min remove it and its left subtree
        //equivalent to returning the trimmed right subtree
        if (root.val < L) {
            root = trimBST(root.right, L, R);
        }
        //if this value is greater than the max remove it and its right subtree
        //equivalent to returning the trimmed left subtree
        else if (root.val > R) {
            root = trimBST(root.left,L,R);
        }
        //otherwise trim the right and left subtrees
        else {
            root.right = trimBST(root.right,L,R);
            root.left = trimBST(root.left,L,R);
        }
        return root;
    }
}
```

Problem Approach for Level 3 problem

No recursion this time, a queue is generally better for level order traversal

1: Create a list that stores the traversal, add the root to the list, add the roots children to the queue

2: While the queue is not empty, pop off a node and add it to the list. Enque all the children of this node

Java Code for level 3 problem

```
class Solution {
    public List<List<Integer>> levelOrder(Node root) {
        //create a queue for the nodes
        Queue<Node> q = new LinkedList<Node>();
        q.add(root);
        //create a map to let us know what depth each node is on
        HashMap<Node, Integer> dist = new HashMap<Node,Integer>();
        dist.put(root,1);
        List<List<Integer>> ret = new ArrayList<List<Integer>>();
        while(q.peek() != null){
            Node curr = q.remove();
            if (ret.size() < dist.get(curr)) {
                ret.add(new ArrayList<Integer>());
            }
            //get the list associated with this depth and add this number to the list
            ret.get(dist.get(curr)-1).add(curr.val);
            //enque all children
            for (Node n: curr.children) {
                q.add(n);
                dist.put(n,dist.get(curr)+1);
            }
        }
        return ret;
    }
}
```

Problem Approach for Level 3b problem

Observation 1: all subtrees of a binary search tree are themselves a binary search

Use recursion!

1: A tree with 0 nodes is a BST

2: Otherwise, for a tree to be a BST its left and right are both BSTs

3: if they are, check to see if the root is bigger than its right child and smaller than its left child. If so it is a BST

Problem: This is NOT good enough to say it is a BST! You have to use the max/min of the right/left subtrees

3(corrected): if they are, check to see if the root is bigger than the MIN of the right subtree, and smaller than the MAX of the left subtree

Java Code for level 3b problem (Part 1)

Helper functions to find the min and max of a subtree

```
class Solution {
    public long maxNode(TreeNode root) {
        if (root == null) {
            return Long.MIN_VALUE;
        }
        if (root.right == null) {
            return root.val;
        }
        return maxNode(root.right);
    }
```

```
public long minNode(TreeNode root) {
    if (root == null) {
        return Long.MAX_VALUE;
    }
    if (root.left == null) {
        return root.val;
    }
    return minNode(root.left);
}
```

Java Code for level 3b problem (Part 2)

```
    public boolean isValidBST(TreeNode root) {
        if (root == null) {
            return true;
        }
        else {
            return isValidBST(root.right) && isValidBST(root.left)
                && (root.val < minNode(root.right))
                && (root.val > maxNode(root.left));
        }
    }
}
```

Problem Approach for Level 4 problem

Observation 1: If a leaf node has some value other than 1, then it needs to either give to or take from its parent

Make all the parents of the leaf nodes distribute the necessary coins to their children (even if they don't have enough!)

1: A null tree requires 0 moves

2: Otherwise, if the a node is a parent to leaf nodes, it has to distribute the apporpriate amount to its children

3: If a node is not a parent to leaf nodes, then it must wait for its children to handle their children, and then it will distribute the correct amount of coins to them

Java Code for level 4 problem (Part 1)

Helper function to determine if a node is a leaf or not

```
public boolean isLeaf(TreeNode root) {
    if (root == null) {
        return false;
    }
    return root.right == null && root.left == null;
}
```

Java Code for level 4 problem (Part 2)

```java
public int distributeCoins(TreeNode root) {
    if (root == null) {
        return 0;
    }
    else if (isLeaf(root)) {
        //parents handle their children so leaf nodes don't do anything
        return 0;
    }
    else {
        int count = 0;
        //if you have leaf children, then give them coins
        if (isLeaf(root.right)) {
            root.val += (root.right.val-1);
            count += Math.abs(root.right.val-1);
        }
        else {
            //otherwise count the moves it takes your children
            //to distribute their coins
            count += distributeCoins(root.right);
            if (root.right!=null) {
                //distribute coins to your children
                root.val += (root.right.val-1);
            }
        }
```

Handling the right subtree

\+ Base cases

Java Code for level 4 problem (Part 3)

Handling the left subtree

```
            if (isLeaf(root.left)) {
                root.val += (root.left.val-1);
                count += Math.abs(root.left.val-1);
            }
            else {
                count += distributeCoins(root.left);
                if (root.left!=null) {
                    root.val += (root.left.val-1);
                }
            }
            return count + Math.abs(root.val-1);
        }
    }
}
```

Next Week: Greedy Algorithms