Section Handout 7

Problem One: Binary Search Tree Warmup!

Binary search trees have a ton of uses and fun properties. To get you warmed up with them, try working through the following problems.

First, draw three different binary search trees made from the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9. What are the heights of each of the trees you drew? What's the tallest BST you can make from those numbers? How do you know it's as tall as possible? What's the shortest BST you can make from those numbers? How do you know it's as short as possible?

Take one of your BSTs. Trace through the logic to insert the number 10 into that tree. Then insert 3½. What do your trees look like?

Problem Two: Walking Through the Trees

Write iterative functions to search a BST for a value and to insert a new value into a BST. Then compare what you wrote to the recursive implementations of those functions. Which ones seem cleaner?

Problem Three: The Ultimate and Penultimate Values

Write a function

```
Node* biggestNodeIn(Node* root);
```

that takes as input a pointer to the root of a (nonempty) binary search tree, then returns a pointer to the node containing the largest value in the BST. What is the runtime of your function if the tree is balanced? If it's imbalanced? Then, write a function

```
Node* secondBiggestNodeIn(Node* root);
```

that takes as input a pointer to the root of a BST containing at least two nodes, then returns a pointer to the node containing the second-largest value in the BST. Then answer the same runtime questions posed in the first part of this problem.

Fun Fact: This first algorithm is how the `Set<T>::first()` function works.

Problem Four: A Problem of Great Depth and Complexity

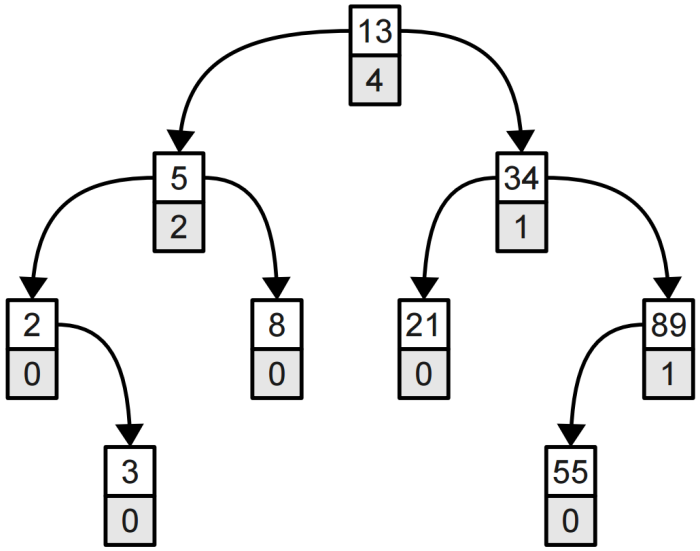
Write a function

```
int heightOf(Node* root);
```

that returns the height of the given tree. By convention, an empty tree has height -1. Then talk about the big-O time complexity of your solution.

Problem Five: Order Statistic Trees

An *order statistic tree* is a binary search tree where each node is augmented with the number of nodes in its left subtree. For example, here is a simple order statistic tree:



Suppose that you have the following struct representing a node in an order statistic tree:

```
struct Node {
      int value;
      int leftSubtreeSize;
      Node* left;
      Node* right;
};
```

Write a function

```
Node* kthNodeIn(Node* root, int k);
```

that accepts as input a pointer to the root of the order statistic tree, along with a number *k*, then returns a pointer to the *k*th-smallest node in the tree (zero-indexed). If *k* is negative or at least as large as the number of nodes in the tree, your function should return `nullptr` as a sentinel. Then, analyze the time complexity of your solution.

Problem Six: Custom Comparators

In Assignment 4, you saw the following `struct`, which represents a voting state:

```
struct State {
    string name;        // The name of the state
    int electoralVotes; // How many electors it has
    int popularVotes;   // The number of people in that state who voted
};
```

In lecture, you saw how to overload the `<` operator to make it possible to store your own custom types as keys in a `Map` and as values in a `Set`. Why was it necessary to do this? What four rules must your function `operator<` obey? Based on that, go and write an `operator<` for the `State` type.

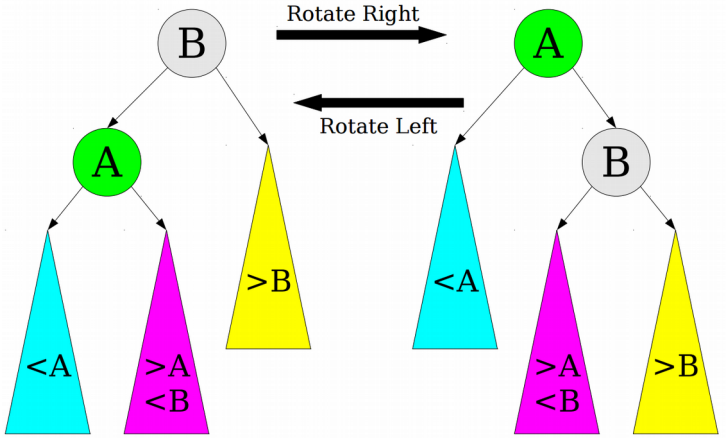
Problem Seven: Freeing Trees Efficiently

In lecture, we saw that you can use the following recursive function to deallocate all the memory used by a binary search tree:

```
void freeTree(Node* root) {
    if (root == nullptr) return;
    freeTree(root->left);
    freeTree(root->right);
    delete root;
}
```

(What kind of tree traversal is this?) The problem with this code is that if you have a highly degenerate tree, say, one that's essentially a gigantic linked list, the recursive depth can get pretty high, so high in fact that it can cause a stack overflow for a sufficiently large tree.

Here's another algorithm you can use to free all the nodes in a tree in O(n) time, using no recursion at all. This algorithm, which I first heard from a friend who now works at a self-driving car company, is based on the idea of *tree rotations*. A ***tree rotation*** is a way of reorganizing the nodes in a binary search tree that changes the tree's shape, but maintains the fact that it's still a binary search tree. There are two kinds of rotations, left rotations and right rotations, which are illustrated here:



Here's the algorithm for deleting all the nodes in the tree. First, imagine the root node has no left child. In that case, we can just deallocate the root and then proceed to clean up the right subtree. Otherwise, the root has a left child. So let's do a single right rotation, moving more of the nodes in to the right subtree, and repeat. Eventually, we'll munch up all of the nodes in the BST, and since there's no recursion involved here, we use only O(1) auxiliary space. Impressively, this still runs in time O(n).

Implement this algorithm.

Problem Eight: Counting BSTs

Write a function

```
int numBSTsOfSize(int n);
```

that takes as input a number n, then returns the number of differently-shaped binary search trees you can make out of n elements.

Problem Nine: Building BSTs Directly

Our first lecture on binary search trees included a mystery function `buildBSTFrom` that, given a set of numbers, produced a BST out of those numbers. This question explores how that function works.

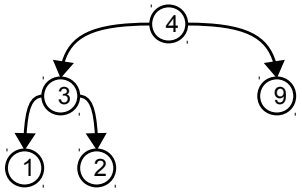
Imagine that you have a sorted list of n numbers stored in a `Vector`. Write a function

```
Node* buildBSTFrom(const Vector<int>& elems);
```

that constructs and returns a ***balanced*** binary search tree containing the elements stored in the list given by `elems`. See if you can find a way to get it to run in time $O(n)$, where n is the number of elements in the `Vector`.

Problem Ten: Checking BST Validity

You are given a pointer to a `Node` that is the root of some type of binary tree. However, you are not sure whether or not it is a binary *search* tree. That is, you might get a tree like the one shown to the right, which is a binary tree but not a binary search tree. Write a function




```
bool isBST(Node* root);
```

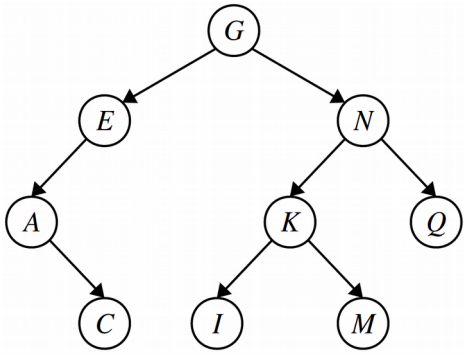
that, given a pointer to the root of a tree, determines whether or not the tree is a legal binary search tree. You can assume that what you're getting as input is actually a tree, so, for example, you won't have a node that has multiple pointers into it, no node will point at itself, etc.

As a hint, think back to our recursive definition of what a binary search tree is. If you have a node in a binary tree, what properties must be true of its left and right subtrees for the overall tree to be a binary search tree? Consider writing a helper function that hands back all the relevant information you'll need in order to answer this question.

Problem Eleven: Deleting from a BST

We didn't talk about how to delete nodes out of a BST, and that's for a good reason – it's surprisingly challenging! Let's suppose you want to delete a node out of a BST. There are three cases to consider:

1. The node is a leaf. In that case, it's really easy to delete, so we just go and delete it.
2. The node has exactly one child. In that case, we delete the node and "replace" it with that one child by updating the node's parent so that it points directly at the single child.
3. The node has two children. In that case, we do the following. Suppose we want to delete the node containing x. Find the node with the largest value in x's left subtree. Copy the value from that node and overwrite the value x. Then, go and delete that new node instead of x.



Trace through this algorithm by hand on the tree to the left, deleting *C*, then *E*, then *N*, then *M*. Then, implement a function

```
void removeFrom(Node*& root, int value);
```

that removes the specified value from the given BST, if that value exists. Finally, discuss the runtime of the algorithm you implemented as a function of the height of the tree.